2016-09

Multiprime Blum-Blum-Shub pseudorandom number generator

Shrestha, Bijesh

Monterey, California: Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

MULTIPRIME BLUM-BLUM-SHUB PSEUDORANDOM NUMBER GENERATOR

by

Bijesh Shrestha

September 2016

Thesis Advisor:	Pantelimon Stanica
Thesis Co-Advisor:	Thor Martinsen

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704–0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 2016	Master's Thesis 09-28-2015 to 09-17-2016

4. TITLE AND SUBTITLE

MULTIPRIME BLUM-BLUM-SHUB PSEUDORANDOM NUMBER GENERATOR

5. FUNDING NUMBERS

6. AUTHOR(S)

Bijesh Shrestha

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

N/A

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release. Distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Blum-Blum-Shub (BBS) is a (probabilistically) secure pseudorandom bit/number generator which outputs a sequence by repeatedly reducing squares modulo the product of two Blum-primes. Our goal for this thesis is to modify the algorithm by using a modulus which is the product of three Blum-primes. We evaluate the effect of this modification using the suite of tests from National Institute of Standards and Technology (NIST). Previous research has evaluated the limit on the number of least important bits that can be extracted per iteration of the BBS algorithm while still maintaining the pseudorandom properties. In this paper, we go beyond the proposed limit and compare the modified BBS with the original BBS using the NIST tests. This paper also discusses the cryptosystem based on the modified BBS as well as the original BBS. We use three metrics for the comparison of performance: the type of tests, the overall performance of sequences against NIST tests, and the time to generate sequences. Our test data shows that both versions performed in a similar manner when subjected to NIST tests. Furthermore, bit generation is significantly faster for sequences generated by taking the last 50 bits or more, while still maintaining pseudorandom properties.

14. SUBJECT TERMS

Pseudorandom number generator, Pseudorandom bit generator, Blum-Blum-Shub, Cryptography, National Institute of Standards and Technology Tests

15. NUMBER OF PAGES 83

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

MULTIPRIME BLUM-BLUM-SHUB PSEUDORANDOM NUMBER GENERATOR

Bijesh Shrestha
Second Lieutenant, United States Army
B.S., University of Alaska, Anchorage, 2015

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN APPLIED MATHEMATICS

from the

NAVAL POSTGRADUATE SCHOOL
September 2016

Approved by: Pantelimon Stanica
Thesis Advisor

Thor Martinsen
Thesis Co-Advisor

Craig W. Rasmussen
Chair, Department of Applied Mathematics

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Blum-Blum-Shub (BBS) is a (probabilistically) secure pseudorandom bit/number generator which outputs a sequence by repeatedly reducing squares modulo the product of two Blum-primes. Our goal for this thesis is to modify the algorithm by using a modulus which is the product of three Blum-primes. We evaluate the effect of this modification using the suite of tests from National Institute of Standards and Technology (NIST). Previous research has evaluated the limit on the number of least important bits that can be extracted per iteration of the BBS algorithm while still maintaining the pseudorandom properties. In this paper, we go beyond the proposed limit and compare the modified BBS with the original BBS using the NIST tests. This paper also discusses the cryptosystem based on the modified BBS as well as the original BBS. We use three metrics for the comparison of performance: the type of tests, the overall performance of sequences against NIST tests, and the time to generate sequences. Our test data shows that both versions performed in a similar manner when subjected to NIST tests. Furthermore, bit generation is significantly faster for sequences generated by taking the last 50 bits or more, while still maintaining pseudorandom properties.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

NPS	Naval Postgraduate School
PRBG	Pseudorandom Bit Generator
TRBG	True Random Bit Generator
CRT	Chinese Remainder Theorem
BBS	Blum-Blum-Shub Pseudorandom Bit Generator
NIST	National Institute of Standards and Technology

THIS PAGE INTENTIONALLY LEFT BLANK

Executive Summary

This thesis concentrates on a modification of the classical Blum-Blum-Shum pseudorandom number generator, which is known to be probabilistically secure. The Blum-Blum-Shum pseudorandom number generator algorithm begins by choosing a modulus n that is the product of two large primes $p, q \equiv 3 \pmod 4$ and a seed x_0, and recursively generates a new bit $x_{n+1} = x_n^2 \pmod n$. We modify the Blum-Blum-Shub pseudorandom bit generator (BBS), by taking the the modulus for the algorithm as a product of three Blum-primes of the same size, a seed x_0 and the same recursion as in the classical case.

We start by comparing the two versions of BBS. In order to do so, we use a Python program to generate a bit sequence of desired length for both versions of the algorithm. The generated sequences are tested for randomness properties using a suite of tests from the National Institute of Standards and Technology (NIST). BBS, in its native form, extracts only the parity bit per iteration of the algorithm. Hence, the number of iterations equals the length of the desired bit sequence. Previous research by Vazirani and Vazirani shows that up to $\log \log N$ bits can be extracted per iteration of BBS, where N is the size of the modulus [1], and the obtained sequence remains secure (probabilistically). Given our chosen length for both versions of the algorithm, this limit is up to nine bits per iteration. In our thesis, we extract a number of bits beyond this theoretical limit and show that the obtained sequence still passes, with high probability, the NIST tests.

For a detailed comparison, we gather samples in eight categories for both versions, where a category corresponds to the number of bits extracted per iteration (the last bit, the last 4 bit, the last 5 bit, the last 10 bit, the last 50 bit, the last 100 bit, and all bits, respectively). Each category is comprised of 200 samples of sequences, approximately one million bits in length.

The generated sequences are tested using the NIST randomness tests, which output numerical values for each sequence, we determine whether or not it satisfies the requirements set forth in the standard. Along with the NIST tests, we also investigate the time required to generate the sequences for all categories. The data for the time requirement focuses on the effect on time as the amount of bits extracted per iteration increases.

For a whole picture, we are interested to see if the randomness properties degrade (or not) as we extract more bits per iteration as well.

The comparison between the original and the modified BBS is done using three metrics. The first comparison is based on the performance in each test, the second comparison is based on the overall performance of each sequence against NIST tests, and the third comparison is based on the time required to generate the sequences by both versions, of roughly one million bits in length, for each category.

In our comparison based on performance in each test, the data suggested that both versions performed with a pass-rate of 95 percent or higher in each of the 14 NIST tests. Looking at the overall performance of each sequence, the data again shows that both versions performed in a similar manner. For both versions, within each category, 70 to 80 percent of sequences passed all of the NIST tests.

For our third comparison, the time required to generate a sequence of one million bits came closer for both versions as we extracted 50 bits or more per iteration. In addition, the time required to generate a one million bit sequence decreased by 90 percent or higher as compared to the initial time for extracting only the parity bit per iteration. During this process, the data, from NIST tests, also suggested that the pseudo-randomness properties for sequences generated did not degrade as the extracted bits per iteration increased.

Additionally, we also introduce a cryptosystem based on the original and the modified BBS. Although not many details are included here (a paper is forthcoming), we make an effort to show how our public key cryptosystem functions.

References

[1] U. V. Vazirani and V. V. Vazirani, "Efficient and Secure Pseudo-Random Number Generation," in *Workshop on the Theory and Application of Cryptographic Techniques.* Heidelberg: Springer, 1984, pp. 193–202.

Acknowledgments

I would like to thank my thesis advisor, Professor Pantelimon Stanica of Department of Applied Mathematics at Naval Postgraduate School, for his continuous support and guidance throughout the process. I am very thankful that he always encouraged me to be a better writer. He always welcomed my questions during or after work hours, and pointed me toward the right direction.

I would also like to thank Naval Postgraduate School Permanent Military Professor Commander Thor Martinsen who served as my thesis co-advisor. I am very grateful for his guidance, encouragement and valuable comments on the thesis. He not only co-advised but also acted as a mentor during my writing process with quick and in-depth reviews on the materials that I wrote. His invaluable insights guided me throughout the entire writing process.

I would also like to thank the United States Army Cyber Command for the opportunity to attend this prestigious institution. Finally, I want to thank my family for their love and support that have carried me to where I am today.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

Cryptography is a part of our daily lives. Many digital communications require secrecy and authentication over open channels such as the Internet. Every day, we use the world wide web for activities such as checking the news or looking up food recipes. Encryption is not required during such low risk activities. However, activities such as online banking requires additional safeguards in order to protect against malicious activities, given the fact that our personal and financial information is at stake. These days, there are a myriad of reasons why people want to secure their online activity. Governmental and non-governmental agencies, commercial and non-profit enterprises, as well as the general public all rely on the security, of which cryptography is an intrinsic part. Encryption is the process of transforming data, in the form of plaintext, into ciphertext using a mathematical algorithm. The ciphertext can then be securely transmitted. At the destination, the intended recipient can subsequently perform decryption which in turn transforms the ciphertext back into plaintext.

In modern cryptography, the security of the encryption scheme should not be based on keeping the algorithm secret. Kerckhoffs's principle tells us to assume that the encryption scheme itself is public knowledge, while only the key remains secret [1]. The encryption scheme is supposed to provide perfect secrecy, that is, only the intended parties can encrypt or decrypt the communications. Claude Shannon defined perfect secrecy as a system where the correlation between the ciphertext and the plaintext is not visible, that is, all keys are equiprobable [2]. In many cryptographic systems, the encryption scheme uses an algorithm to produce a string of 1's and 0's that appear random. Such a string is called a pseudorandom bit sequence (it is deterministic). This string is then mathematically combined with the plaintext in order to produce the ciphertext. In order to ensure that the ciphertext does not show any correlation to the plaintext, we need to produce a binary string where the bits do not exhibit any patterns which might help an adversary gain insight into the mathematical algorithm which produced them. This in turn requires a good algorithm.

In order to achieve strong encryption, a random bit sequence is required. A truly random bit sequence does not show any pattern among the generated bits that could in any way help an adversary decrypt the ciphertext. Such random sequences can be obtained by observing

atmospheric noises, radioactive decay, and other natural phenomena, which produce non-deterministic outputs. In cryptography the problem we face is that we are surrounded by only a finite number of random natural phenomena and using these phenomena to generate random bit sequences is slow and inefficient. For a practical encryption process, we require alternative ways of generating such sequences that appear random but can be generated quickly. In particular, we need a deterministic algorithm, which generates pseudorandom bit sequences without any observable patterns.

In this thesis, we will discuss pseudorandom bit generators (PRBG). In particular, we will focus on a PRBG called Blum-Blum-Shub (BBS) and we propose a modification to this original idea. We will generate bit sequences using the original BBS as well as the modified BBS and subsequently subject them to a battery of randomness tests developed by the National Institute of Standards and Technology.

1.1 Background

In this section, we will discuss the characteristics and importance of true and pseudorandomness. An example of true randomness is a fair coin toss. A fair coin toss is a simple and commonly used example when defining unpredictability and the probability of guessing outcomes. When a fair coin is tossed, the probability of getting heads or tails is 50/50. This means that no matter how many times the coin is tossed, the likelihood of correctly predicting the next outcome is equal to the likelihood of making an incorrect prediction. Next, we discuss the true-random bit generator (TRBG) and its properties.

1.1.1 True-Random Bit Generator

A true-random generator makes use of the entropy present in natural phenomena to create a truly random binary sequence [3]. The bits in such a string are produced using a non-deterministic process [4]. However, for cryptographic implementations, true-random bit generators may not be a good fit as they require almost instantaneous access to the random bits while encrypting the data. Though preferred, our access to true-random sequences generated by TRBGs are limited for cryptographic use. Next, we discuss alternatives to the TRBG.

1.1.2 Pseudorandom Bit Generator

A pseudorandom bit generator (PRBG) is a deterministic algorithm which, given a binary input, outputs a random-appearing binary bit sequence [5]. PRBGs are used in many applications, both secure and insecure. In both cases, the output must appear to be random. In non-secure application such as simulations in academic environment, and so on, the user is not concerned whether or not the bit-generation method can be determined from the PRBG output. On the other hand, the PRBGs used for encryption and secure transmission or storage of data must be provably secure. In particular, the bits in the generated sequence need to exhibit the unpredictability of a fair coin toss in order to be cryptographically secure. Pseudorandom bit sequences are generated using a deterministic algorithm, yet their outputs needs to appear as though they were truly random.

The desired properties of a pseudorandom bit generator are:

- The process for generating pseudorandom bits needs to be fast.
- The generated bit sequence should have proportional amounts of 1's and 0's.
- The sequence of 1's and 0's should not follow a pattern. In short, the likeliness of both should be no better than flipping a fair coin.
- The generated bit sequence should pass the suite of various randomness tests.

Various forms of encryption require pseudorandom bit sequences. A simple example is the use of stream ciphers in many cable television broadcasts today. The signal sent from the cable company to the user's cable box is encrypted and decrypted using a stream of pseudorandom bits. The bit by bit encryption/decryption is accomplished by adding (Exclusive-Or) a pseudorandom bit sequence to the data (this if often called a one-time pad). For example, suppose our message is $M = 1100110$. In order to encrypt it, we add a string of random bits, $B = 1011010$ and Exclusive-Or (XOR) denoted as $\oplus$. This produces a ciphertext C. Mathematically, $C = M \oplus B = 1100110 \oplus 1011010 = 0111100$. Decryption is carried out by adding B to the ciphertext. Hence, the decrypted message, $M = C \oplus B = 0111100 \oplus 1011010 = 1100110$.

The next chapter discusses the theory behind the original Blum-Blum-Shub pseudorandom bit generator (BBS) as well as the number theory behind the idea in order to aid the reader in understanding the algorithm. We also discuss the associated cryptosystem based on both

versions of BBS. In Chapter 3, we go over our methodology for modifying the original BBS and generating pseudorandom bit sequences from both generators, which will be tested using a suite of tests by the National Institute of Standards and Technology [6]. This chapter explains what each test does and how to read the results. We will also lay the groundwork on how we compare the results that we get from the tests and how we interpret the results. Chapter 4 focuses on the results of the tests. Finally, Chapter 5 offers a conclusion along with recommendations for future work.

CHAPTER 2:
Blum-Blum-Shub

Chapter 2 explores the theory behind BBS and its associated cryptosystem. We also discuss the proposed modification of BBS and address a new cryptosystem based on this modified generator.

2.1 Blum-Blum-Shub Pseudorandom Bit Generator

Lenore Blum, Manuel Blum and Michael Shub proposed the Blum-Blum-Shub pseudorandom number generator, and the original paper was published in 1986 [7]. In order to generate pseudorandom bits, the BBS algorithm requires large prime numbers called Blum-primes, which are prime numbers p such that $p \equiv 3 \pmod 4$ [8], [7].

Definition 1. *The Blum-Blum-Shub pseudorandom bit generator (BBS) algorithm is as follows [5]:*

- *Generate two large Blum-prime numbers, p and q such that* $\gcd(p, q) = 1$
- *Let* $n = p \cdot q$
- *Choose a random seed,* $x_0 \in \{1, 2, \ldots, n-1\}$
- *Let* $x_i \equiv x_{i-1}^2 \pmod n$ *and* $z_i = x_i \pmod 2 = Parity\ (x_i)$
- *The output sequence is* $Z = (z_i)_{i \geq 1}$.

Example 1. *The example below will demonstrate how these pseudorandom bit sequences are generated via BBS:*

- *Let,* $p = 7 \equiv 3 \pmod 4$ *and let* $q = 19$, *which is* $3 \pmod 4$. *Then,* $n = p \cdot q = 7 \cdot 19 = 133$
- *Choose* x_0 *(a random seed)* $= 100$
- *Then* $x_1 = 100^2 \pmod{133} = 25$
- $x_2 = 25^2 \pmod{133} = 93$
- $x_3 = 93^2 \pmod{133} = 4$
- $x_4 = 4^2 \pmod{133} = 16$
- $x_5 = 16^2 \pmod{133} = 123$, *etc.*

- z_1 = *the parity of* $x_1 = 1$. *Similarly,* $z_2 = 1$, $z_3 = 0$, $z_4 = 0$, $z_5 = 1$, *and so on.*
- *Hence, the pseudorandom bit sequence is* $11001\ldots$. *The sequence obviously repeats after as many as the modulus number of steps (sometimes, fewer).*

2.2 BBS as a Cryptosystem

In a public key cryptosystem, we have two separate keys for encryption and decryption. One key is called a public key, and the other is called a private key. Each individual will have a unique public and private key. As the name suggests, the public key is either published or distributed in clear. On the other hand, the private key is kept secret. For example, if Bob wants to send a message to Alice, he encrypts his message using Alice's public key and then send the encrypted message to her. Only Alice, who possesses the corresponding private key, can decrypt and read the original message. Public key cryptosystem relies on these premises.

In this section, we demonstrate how BBS could be used as a secure public key cryptosystem. Suppose that Alice and Bob want to communicate to each other. Assume Alice is the recipient and Bob is the sender. Alice, the recipient, will generate two large Blum-primes p and q such that the modulus $n = p \cdot q$. Here, p and q are kept secret and n is public. The security of the cryptosystem relies on the fact that anyone can encrypt using n as a modulus, but the decryption is only possible if the prime factors p and q are known. In [9], Jeremy Booher describes the methodology for finding the square roots that makes the decryption possible.

The BBS cryptosystem works as follows:

1. Suppose Bob needs to encrypt the message $M = m_1, m_2, \ldots, m_k$, $k < n$. The encryption process is performed as follows:
   - Choose a random seed, $x_0 \in \{1, \ldots, n-1\}$.
   - $x_i \equiv x_{i-1}^2 \pmod{n}$, $1 \leq i \leq k+1$.
   - $z_i = \text{Parity}\ (x_i)$, $1 \leq i \leq k$.
   - The pseudorandom bit sequence generated is $Z = (z_1, z_2, z_3, \ldots, z_k)$.
   - Compute the ciphertext $C = M \oplus Z = (c_1, c_2, c_3, \ldots, c_k)$, which is sent to Alice along with the integer x_{k+1}. Note that $x_{k+1} = x_k^2 \pmod{n}$ is an integer, which

will allow Alice to decrypt.

2. Alice receives the ciphertext sent by Bob and decrypts it using the factors p and q to find the square roots. Decryption is done as follows:
   - Alice receives the ciphertext $c_1, c_2, c_3, \ldots, c_k$, along with the extra information x_{k+1}. She takes x_{k+1} to calculate (see the next step) the pseudorandom bit sequence $Z' = (z_k, z_{k-1}, \ldots, z_2, z_1)$.
   - The residue x_k modulo n is calculated using the Chinese Remainder Theorem:

$$x_k \equiv x_{k+1}^{\frac{p+1}{4}} \pmod{p}$$
$$x_k \equiv x_{k+1}^{\frac{q+1}{4}} \pmod{q}$$

   - The parity of the x_k modulo n is the last bit z_k of the pseudorandom sequence used to encrypt the original message.
   - The remaining bits of the pseudorandom sequence is calculated by repeating the process. At the end, the extracted bits in pseudorandom sequence $Z' = (z_k, z_{k-1}, \ldots, z_2, z_1)$ are reversed such that the original pseudorandom bit sequence, $Z = (z_1, z_2, z_3, \ldots, z_k)$ is recovered.
   - The original message is found by computing $M = C \oplus Z$.

Below is an example of a message being sent using the BBS cryptosystem.

Example 2. *Alice picks two Blum-primes $p = 7$ and $q = 19$ such that the modulus $n = 7 \cdot 19 = 133$. She keeps p and q private and sends $n = 133$ to Bob. The encryption is done as follows:*

- *Assume that the plaintext message is $M = 1010$.*
- *Bob chooses a random seed, $x_0 = 100$ such that $1 < x_0 < N$.*
- $x_1 = x_0^2 \pmod{N} = 100^2 \pmod{133} = 25$.
- $x_2 = x_1^2 \pmod{N} = 25^2 \pmod{133} = 93$.
- $x_3 = x_2^2 \pmod{N} = 93^2 \pmod{133} = 4$.
- $x_4 = x_3^2 \pmod{N} = 4^2 \pmod{133} = 16$.
- $x_5 = x_4^2 \pmod{N} = 16^2 \pmod{133} = 123$.
- *Bob will compute the parity bits such that $z_1 = Parity\ (x_1) = 1$. Similarly, $z_2 = 1$, $z_3 = 0$ and $z_4 = 0$. Hence, the pseudorandom bit sequence is $Z = 1100$.*

- *The ciphertext is then* $C = M \oplus Z = 1010 \oplus 1100 = 0110$.
- *The message sent to Alice is ciphertext C along with* x_5 *appended at the end of the message. For the purpose of this example, the ciphertext sent is* 0110(123). *(Note:* $x_5 = 123$ *is used as the decimal representation instead of its binary representation).*

Once Alice receives the message, the decryption is done as follows:

- *Alice uses the Chinese Remainder Theorem to solve for* x_4,

$$x_4 \equiv 123^{\frac{7+1}{4}} \pmod 7$$
$$x_4 \equiv 123^{\frac{19+1}{4}} \pmod{19},$$

  and she finds the value $x_4 = 16$.
- *The last bit of the pseudorandom sequence is* $z_4 = Parity(x_4) = Parity(16) = 0$.
- *Alice will repeat the process until all* 4 *bits of pseudorandom sequence used for encryption has been recovered.*
- $Z' = (z_4, z_3, z_2, z_1) = 0011$ *(we removed commas, for convenience). Hence,* $Z = 1100$ *is obtained by reversing (mirror image) the bits of* Z'.
- *The decrypted message then equals* $M = 0110 \oplus 1100 = 1010$.

2.3 Elementary Number Theory

The security of BBS is based on the difficulty of solving the quadratic residuosity problem. As long as appropriate primes are selected and at most $loglogN$ bits are output Junod and Ding have shown that the generator will remain secure. [5], [10]. We have thus far discussed how the algorithm generates the bit sequence of desired length, but must now also discuss some number theory in order to understand the security aspects of the generator. Note that the logarithm base in this paper is 2.

Definition 2 ([5]). *Let* $n \in \mathbf{N}$ *and* $\mathbf{Z_n^*}$ *be the set of positive integers modulo n. Then* $a \in \mathbf{Z_n^*}$ *is called a quadratic residue modulo n if there exists* $b \in \mathbf{Z_n^*}$ *such that:*

$$a \equiv b^2 \pmod n.$$

The set of quadratic residues modulo n is denoted by QR_n. *Furthermore, the set of quadratic*

non-residues is defined as

$$QNR_n := \mathbf{Z}_\mathbf{n}^* \setminus QR_n.$$

Definition 3 ([5]). *Let p be an odd prime and* $\mathbf{Z}_\mathbf{p}^*$ *denote the set of positive integers modulo p. For* $a \in \mathbf{Z}_\mathbf{p}^*$*, the Legendre symbol* $\left(\frac{a}{p}\right)$ *is defined as:*

$$\left(\frac{a}{p}\right) = \begin{cases} 0 & \text{if } p \setminus a \\ 1 & \text{if } a \in QR_p \\ -1 & \text{if } a \notin QR_p. \end{cases}$$

Theorem 1. *Let p be an odd prime, and let* $a \in \mathbf{Z}_\mathbf{p}^*$*, Then*

$$\left(\frac{a}{p}\right) \equiv a^{\frac{p-1}{2}} \pmod{p}.$$

Proof. We now follow closely Junod's proof of Theorem 1 from [5]. Let $a \in QR_p$, and $a \equiv b^2 \pmod{p}$ for some $a \in \mathbf{Z}_\mathbf{p}^*$. Using Fermat's Little Theorem,

$$a^{\frac{p-1}{2}} \equiv (b^2)^{\frac{p-1}{2}} \equiv 1 \pmod{p}.$$

Now, assume $a \notin QNR_p$. Let g be a generator of $\mathbf{Z}_\mathbf{p}^*$. Then $a \equiv g^t \pmod{p}$, where t is odd. Otherwise, $a \equiv g^t \equiv g^2 s = (g^s)^2$. Hence, for $t = 2s + 1$,

$$a^{\frac{p-1}{2}} \equiv (g^t)^{\frac{p-1}{2}} \equiv (g^{2s+1})^{\frac{p-1}{2}} \equiv (g^{2s})^{\frac{p-1}{2}} \cdot g^{\frac{p-1}{2}} \equiv g^{\frac{p-1}{2}}.$$

Here, $(g^{\frac{p-1}{2}})^2 = 1$. Thus, $g^{\frac{p-1}{2}} = 1$ or -1. Since, g is a generator of $\mathbf{Z}_\mathbf{p}^*$, the order of g is $p - 1$, and $g^{\frac{p-1}{2}} = -1$. □

Theorem 1 is very helpful as it lets us determine whether an element is a quadratic residue of the finite field. It is also known that if p is an odd prime, where $p \in \mathbf{Z}_\mathbf{p}^*$, then the number of quadratic residue equals to the number of quadratic non residues [5].

Theorem 2. *Let p be an odd prime and* $a, b \in \mathbf{Z}$*. Then*

$$\left(\frac{a}{p}\right) \cdot \left(\frac{b}{p}\right) = \left(\frac{ab}{p}\right)$$

Proof. We again use [5]. By Theorem 1,

$$\left(\frac{a}{p}\right) \cdot \left(\frac{b}{p}\right) = a^{\frac{p-1}{2}} \cdot b^{\frac{p-1}{2}} = (a \cdot b)^{\frac{p-1}{2}} \pmod{p} = \left(\frac{ab}{p}\right).$$

□

Next, we discuss the Jacobi symbol, which is analogous to the Legendre symbol for composite moduli. This concept is particularly important when dealing with BBS because the modulus n is not just a prime number but rather a composite number, a product of Blum-primes.

Definition 4 ([5])**.** *Given an odd integer n with a prime factorization, such that* $n = \prod_i p_i^{e_i}$ *and* $a \in \mathbf{Z}_\mathbf{p}^*$*, the Jacobi Symbol* $\left(\frac{a}{n}\right)$ *is defined as*

$$\left(\frac{a}{n}\right) := \prod_i \left(\frac{a}{p_i}\right)^{e_i}.$$

Certainly (see [5], for example), if $n = p \cdot q$, where p, q are distinct primes, then exactly half of the elements have the Jacobi symbol +1 and other half have the Jacobi symbol of -1. These sets are denoted as $\mathbf{Z}_n^*(+1)$ and $\mathbf{Z}_n^*(-1)$, respectively.

Theorem 3. *Let n be an odd integer, and let* $a, b \in \mathbf{Z}$*. Then*

$$\left(\frac{ab}{n}\right) = \left(\frac{a}{n}\right) \cdot \left(\frac{b}{n}\right).$$

Proof. The proof is well known and given in most elementary number theory books. Since we mostly used [5] we provide his approach. Using Definition 4, we get

$$\left(\frac{ab}{n}\right) = \prod_i \left(\frac{ab}{p_i}\right)^{e_i}.$$

Using Theorem 2, we get

$$\prod_i \left(\frac{ab}{p_i}\right)^{e_i} = \prod_i \left(\frac{a}{p_i}\right)^{e_i} \cdot \prod_i \left(\frac{b}{p_i}\right)^{e_i} = \left(\frac{a}{n}\right) \cdot \left(\frac{b}{n}\right).$$

□

As discussed earlier, the decryption in the BBS cryptosystem is possible only if one knows p and q, which is the factorization of n. It is known that a nonzero quadratic residue in a finite field, $\mathbf{Z_p}$, where p is prime, has exactly two square roots [5].

Since BBS uses composite moduli, we will look at the square roots of quadratic residue in a general multiplicative group. For a composite moduli n, where n is an integer, Junod showed that if $a \in QR_n$, then a will have exactly 2^k distinct square roots, where k is the number of distinct prime factors of n [5]. Note that for BBS, where $n = p \cdot q$, we get 4 distinct square roots.

During the decryption process of the BBS cryptosystem, we used the Chinese Remainder Theorem to find the correct square root of the residue. In his paper, Junod proves that for a composite moduli, where $n = p_1, p_2, ..., p_k$, where p_i are distinct prime, any element $a \in \mathbf{Z_n^*}$ is a quadratic residue modulo n if and only if a is the quadratic residue of $\mathbf{Z_{p_i}^*}$ for each prime p_i corresponding to the moduli [5]. This is important because in the BBS cryptosystem, n is known. Thus, even if an adversary knows the modulus n and the last residue x_{n+1}, taking the square root will not help in the calculation of x_n. Hence, during the decryption process, the Chinese Remainder Theorem is used to calculate the unique residue.

To further clarify the uniqueness of the quadratic residues and the use of Blum-primes, we look at following results [5].

Theorem 4. *If p is an odd prime number, then -1 $\in QNR_p$ if and only if p is a Blum-prime.*

Proof. We cite Junod's paper again [5].
Using Theorem 1, we get:

$$\left(\frac{a}{p}\right) \equiv (-1)^{\frac{p-1}{2}} \pmod{p}.$$

Since p is an odd prime, $p \equiv 1 \pmod 4$ or $p \equiv 3 \pmod 4$. Furthermore, $\frac{p-1}{2}$ is odd if and only if $p \equiv 3 \pmod 4$ [5]. □

Theorem 4 answers the question why BBS uses Blum-primes as opposed to just any other primes. Since, $-1 \notin QR_n$, the BBS algorithm will not yield the residue modulo n to be

-1 during any iteration. This is important because if -1 belonged to the quadratic residue modulo n, then each bit after this iteration would yield 1's, resulting in a non-random sequence.

In the original BBS, where the modulus, $n = p \cdot q$, where p and q are Blum-primes, out of the four possible square roots for each element in the quadratic residue, exactly one is in the quadratic residue itself. Junod's paper includes the detailed proof [5].

Next, we propose a modification to the original BBS and discuss the cryptosystem based on this modification.

2.4 Our Modification of BBS

Our goal for this paper is to modify the BBS algorithm with the hope of getting better random sequences when tested against the NIST standards. The pseudorandom binary sequence that we get from the modified BBS is therefore compared to the original BBS for the randomness using the suite of tests from NIST.

Modification: We will randomly generate a new large Blum-prime r, and use the modulus

$$n = p \cdot q \cdot r.$$

We discuss the details on the modification and testing in the next chapter. We begin by discussing the cryptosystem based on the modification.

2.4.1 Modified BBS Cryptosystem: All Blum-Primes

The modified BBS cryptosystem is defined in the same way as the original one. The only difference during the encryption process is that the modulus n, is the product of three Blum-primes. Now, Alice will still keep p, q, r as a private key and send the modulus n to Bob. Bob will use the modulus n in the exact same way as he did before in the original BBS cryptosystem to send the message to Alice.

There is only one added step on Alice's end for decryption. Once Alice receives the ciphertext, say $C = (c_1, c_2, c_3, \ldots, c_k, x_{k+1})$, the decryption is done again using the Chinese

Remainder Theorem but now for the three prime number p, q, and r. Here,

$$\begin{aligned} x_k &= x_{k+1}^{\frac{p+1}{4}} \pmod{p} \\ x_k &= x_{k+1}^{\frac{q+1}{4}} \pmod{q} \\ x_k &= x_{k+1}^{\frac{r+1}{4}} \pmod{r}. \end{aligned}$$

2.5 Modified BBS: Using 2 Blum-Primes and 1 Non Blum-Prime

In this section, we discuss expanding our choices for the primes beyond only Blum-primes. To do so, we choose two large Blum-primes that are congruent to 3 modulo 4, and 1 large prime that is congruent to 1 modulo 4. If Bob sends Alice the encrypted message $C = (c_1, c_2, c_3, \ldots, c_k, x_{k+1})$, then Alice should still be able to decrypt the message using the private key p, q, r and the number x_{k+1}. Below, we demonstrate that the decryption process still works:

- If $p \equiv 1 \pmod 4$ and $q, r \equiv 3 \pmod 4$, then $pq \equiv 3 \pmod 4$.
- For the Blum-prime $r \equiv 3 \pmod 4$,

$$\begin{aligned} x_k^2 &= x_{k+1}^{\frac{r-1}{2}+1} \pmod{r} \\ &= x_{k+1} \pmod{r}. \end{aligned} \tag{2.1}$$

- For pq, we need to show that

$$\begin{aligned} x_k^2 &= x_{k+1}^{\frac{pq-1}{2}+1} \pmod{pq} \\ &= x_{k+1}^{\frac{pq-1}{2}} \cdot x_{k+1} \pmod{pq} \\ &= x_{k+1} \pmod{pq}, \end{aligned} \tag{2.2}$$

  which will follow from

$$x_{k+1}^{\frac{pq-1}{2}} \pmod{pq} \equiv 1 \pmod{pq}.$$

To argue that, observe that

$$\begin{aligned}\frac{pq-1}{2} &= \frac{(p-1)\cdot(q-1)}{2} + \frac{(p-1)}{2} + \frac{(q-1)}{2} \\ &= \frac{\phi(pq)}{2} + \frac{(p-1)}{2} + \frac{(q-1)}{2}.\end{aligned} \tag{2.3}$$

Now,

$$\begin{aligned}x_{k+1}^{\frac{pq-1}{2}} \pmod{pq} &= x_{k+1}^{\frac{\phi(pq)}{2}+\frac{(p-1)}{2}+\frac{(q-1)}{2}} \pmod{pq} \\ &= x_{k+1}^{\frac{\phi(pq)}{2}} \cdot x_{k+1}^{\frac{(p-1)}{2}} \cdot x_{k+1}^{\frac{(q-1)}{2}} \pmod{pq} \\ &\equiv 1 \pmod{pq}.\end{aligned} \tag{2.4}$$

Now, the Chinese Remainder Theorem can be used to recover the bits used for the encryption and subsequently, decrypt the message.

In the next chapter, we will discuss the methodology and the details on how the data is collected.

CHAPTER 3:
Methodology

In Chapter 2, we discussed BBS, the number theory behind the generator, the modification to BBS, as well as the cryptosystems based on both versions. In this chapter, we explain our research methodology, including the code used to generate the pseudorandom sequences, the modification of BBS, the sample gathering, the tests for randomness, and the comparison of our results with previously known ones.

3.1 Codes and Testing

In order to generate the desired bit sequence, we utilized a Python program. A complete copy of the code used in our research can be found in the Appendix.

3.1.1 Description of Python code:

The Python code used in our research is divided into three files:

1. The Blum-prime generator ($PrimeGen.py$) generates a big random prime of desired size when the function is called. The generator program outputs a Blum-prime. Appendix A.2 lists a complete copy of the code.
2. The $Rantest.py$ program contains a suite of NIST randomness tests. The program consists of 14 different types of statistical tests. Appendix A.3 lists the complete copy of the code.
3. The $MainBBS.py$ program is the working interface that employs the BBS method to generate the desired length of pseudorandom sequences. It does so by first calling the functions from $PrimeGen.py$ to obtain the big Blum-primes needed to generate the sequence and subsequently tests the sequence by calling the $Randtest.py$ program. Appendix A.1 lists the complete copy of the code.

The test results were compiled in an Excel file. Next, we talk about our method to generate the bit-strings for both the original and the modified BBS, respectively.

3.1.2 Generating BBS Bit-Strings

Our code generates two large Blum-primes p and q of 256 bits size and we take the modulus to be

$$n = p \cdot q.$$

The program generates a big Blum-prime of size 300 bits, which is used as a seed. The larger size was purposely chosen to guarantee that the seed is coprime to the modulus. For our tests, we generate sequences of size roughly one million bits. The bit sequences are then subjected to a suite of NIST tests. The data is collected for each type of tests and sorted into two categories. The first category shows the number of strings that failed a particular test, and the second category shows the number of strings that failed one or more tests. These two processes are done for both the original and the modified version of BBS.

3.1.3 Generating Modified BBS Bit-Strings

In the original BBS, $n = p \cdot q$, where p and q are randomly generated large Blum-primes. For the modified version, we will introduce a third, call it r, randomly generated large Blum-prime of size 256 bits, and we use the following modulus for our BBS modification

$$n = p \cdot q \cdot r.$$

The bits are generated in the similar way as in the original BBS. The only difference is the modulus n. In the original BBS, $n = p \cdot q$ and in modified BBS, $n = p \cdot q \cdot r$. The obtained (presumably) pseudorandom bit sequence is run through the NIST tests and the results are recorded to compare the two versions of BBS.

3.1.4 Getting Test Samples

As explained in Chapter 2, BBS generates the sequence by taking the parity bit in each step, which generates a cryptographically secure pseudorandom bit sequence. Umesh V. Vazirani and Vijay V. Vazirani [11] proved that it is possible to take up to k bits, where

$$k \leq \log \log N,$$

where N denotes the size of the modulus, and the sequence remains cryptographically secure. In this paper, we not only want to test the effects of introducing a third large Blum-prime but also want to see the results from the randomness tests when more than one parity bit is taken in each steps of the algorithm. Some research has been done on the security of BBS when taking the last 2 bits in each iteration, but little has been investigated on the effects when more than the parity bit is taken in each interval [11]. In order to obtain enough data to make proper comparisons, we have taken seven different types of samples, where the length of bit-sequences is roughly around one million bits. The first seven samples takes the last 1, 3, 4, 5, 10, 50, and 100 bits respectively to generate the bit-sequences of length roughly one million bits. The last sample uses all of the bits from each iteration to generate the bit-sequence of length of roughly one million bits.

In this paper, the modulus is of different size for the original and the modified BBS. The original BBS has modulus of size $N = 2^{512}$ (two large Blum-primes of size 256 bits each so that $n = p \cdot q$) and the modified version has a modulus of size $N = 2^{768}$ (three large Blum-primes of size 256 bits each so that $n = p \cdot q \cdot r$).

3.1.5 Analyzing the Test Results

Our motivation for this research is to analyze the security, randomness and time requirement of the modified BBS when compared to the original BBS. We have broken this part of our research into the following categories:

1. Test type: Here, we have a total of 14 different tests for randomness. First, we look at the performance of sequences against each test and see if the sequences performed better or worse in one of the tests. This will give us a broader view on the performance of both versions. It will help analyze the impact of modifications based on the type of a particular test.
2. Overall performance: Next, we look at the number of sequences that passed all the tests. This will show the performance of each sequence that is being tested. We will also look at the sequences that failed one or more tests. This will be further subdivided based on the number of failed tests.
3. Time requirement: For this comparison, we run both versions of BBS on an Apple Macbook Air with the following specifications:

- Operating System: OS X El Capitan
- Version: 10.11.4
- Type: Macbook Air (13 inch, Early 2015)
- Processor: 1.6 GHz Intel Core i5
- Memory: 4 GB 1600 MHz DDR3
- Graphics: Intel HD Graphics 6000 1536 MB

We record the time taken to generate 200 sequences for the last 1, 3, 4, 5, 10, 50, and 100 bits, respectively.

3.2 Tests for Randomness

As discussed earlier, the generated bit strings from the pseudorandom bit generators are tested to see if they adhere to the desired randomness properties using the suite of tests from NIST [6]. For our research we use the following tests to evaluate the randomness properties of both versions of BBS:

1. The Frequency (Monobit) Test
2. Frequency Test within a Block
3. The Runs Test
4. Tests for the Longest-Run-of-Ones in a Block
5. The Binary Matrix Rank Test
6. The Discrete Fourier Transform (Spectral) Test
7. The Non-overlapping Template Matching Test
8. The Overlapping Template Matching Test
9. Maurer's "Universal Statistical" Test
10. The Linear Complexity Test
11. The Approximate Entropy Test
12. The Cumulative Sums (Cusums) Test
13. The Random Excursions Test
14. The Random Excursions Variant Test

As discussed in Chapter 2, the NIST tests are performed in Python. We use the code as-is and without making any changes, since the tests are universally accepted standards. This

includes the block size and number of blocks, which we will discuss later in the chapter. The statistical tests performed determine the (confidence in the) randomness of a sequence. Since, randomness is a probabilistic feature, it can be interpreted in terms of probability. Here, the tests are designed to test a null hypothesis H_0, which is the hypothesis that the sequences are random. Along with the null hypothesis, the alternative hypothesis H_a is that the sequences are not random. For each individual test performed we either accept the null hypothesis or reject it. This is done based upon comparing the p-value generated during each test with a predetermined standard. As defined by NIST, the p-values represent the probability that a perfect pseudorandom bit generator would generate a less random sequence compared to the sequence that is tested [6]. According to NIST documentation, a significance (benchmark) value of 0.01 is chosen. Here, a significance value of 0.01 means that if 100 non-random sequences are tested, then 1 out of 100 is expected to be rejected. If the p-value ≥ 0.01, then the sequence is considered random with a confidence of 99% [6].

Below, we discuss the key ideas for the mentioned 14 different NIST tests. For detail explanations, please refer to [6]. The explanations below are taken from the source to highlight the key ideas.

3.2.1 The Frequency (Monobit) Test

This test checks if there are an even number of ones and zeros in the binary string. A sequence is considered random/balanced, if it contains an even number of ones and zeros. In order to test whether or not the sequences are balanced, we perform statistical hypothesis testing. The test outputs a p-value, which determines the randomness of the given sequence.

- If the p-value is < 0.01, then the sequence is considered non-random.
- If the p-value is ≥ 0.01, then it is deemed random.

The test has a recommended input size of at least 100 bits for a successful result [6]. For our tests, the input size of the binary sequence is roughly one million bits in length.

3.2.2 Frequency Test within a Block

According to [6], this test examines the frequency of ones and zeros within blocks of size M. In order to do so, the block size must be a divisor of the sequence length. If this is not

the case, the number of blocks of size M is taken to be $N = \left\lfloor \frac{n}{M} \right\rfloor$ and the bits following the last block are discarded. As before, the number of ones and zeros in each block should agree in order for the sequence to appear random. The NIST recommended standards for this test are:

- The input size, $n \geq 2^{100} - 1$ (more than 100 bits)
- The block size, $M \geq 20$
- The block size, $M \geq (0.01)n$
- The number of blocks, $N \leq 100$

The test outputs a p-value, which determines the randomness of the given sequence. If the p-value of the test is ≥ 0.01 then the sequence is considered random [6].

3.2.3 The Runs Test

According to [6], given a sequence, a *run* is defined as an uninterrupted maximal subsequence of identical bits. A *run* of length k, therefore has k identical bit with bits of opposite value at either ends. For example, a run of ones is preceded and followed by zeros. The likelihood of having a one in a truly random sequence is $\frac{1}{2}$. Therefore, the likelihood of having a run of length k is $\frac{1}{2^k}$. In order to ensure that the sequence appears random, there should be no discernible pattern of *runs* of particular lengths. The NIST test examine the sequence to ensure that this is in fact the case. The test outputs a p-value that determines the randomness of the given sequence. A sequence with the p-value of ≥ 0.01 is considered random. The recommended size for the test input is $n \geq 100$ bits [6]. For our test, we use an input size of approximately one million bits.

3.2.4 Tests for the Longest-Run-of-Ones in a Block

This test is similar to the Runs test. Reference [6] explains that this test checks for the longest run of ones in an M-bit block. As before, we alter the length of the sequence n to ensure it is a multiple of the block size. This test is used to determine whether the longest run of ones is similar to what we could expect in a random sequence. NIST guidelines on the length of string n and length of the block sizes and p-value are as follows:

- Minimum length $n = 128$ bits then $M = 8$ bits

- Minimum length $n = 6272$ bits then $M = 128$ bits
- Minimum length $n = 750,000$ bits then $M = 10^4$ bits
- If the p-value of ≥ 0.01, the sequence is considered random [6].

3.2.5 The Binary Matrix Rank Test

This test searches for any linear dependence among fixed-length substrings of the original sequence of length n. It is done by creating matrices from those sub-strings and statistically analyzing them for any kind of linear dependence. The matrices created are of size $M \times Q$, where the number of blocks $N = \left\lfloor \frac{n}{M \cdot Q} \right\rfloor$ [6]. The test outputs a p-value, which determines the randomness of the given sequence. The NIST standard for this test is:

- If the p-value is < 0.01, then the sequence is considered non-random.
- If the p-value is ≥ 0.01, then the sequence is deemed random.

For the input size of our sequences, $M = Q = 32$ is recommended [6].

3.2.6 The Discrete Fourier Transform (Spectral) Test

According to [6], the purpose of this test is to detect the presence of repetitive patterns in the given binary sequence. For the test, the 0's and 1's of the given binary bit sequence $S_n = \epsilon_1, \epsilon_2, \ldots, \epsilon_n$, where $\epsilon_i = 1$ or 0, $\forall i = 1, 2, \ldots, n$ is converted into a new sequence $X = x_1, x_2, \ldots, x_n$, where $x_i = 2\epsilon_i - 1$. Next, the Discrete Fourier transform (DFT) is applied on X producing complex variables representing the periodic components of the sequence of the bits at various frequencies. The test compares whether the peaks that exceeds 95% threshold are significantly different than that of 5%. The test outputs a p-value that determines the randomness of the given sequence. As before, if the p-value is ≥ 0.01, then this NIST tests considers the sequence pseudorandom. The input recommendation is a minimum of 1000 bits [6]. For our research, we use an input size of approximately one million bits for this test.

3.2.7 The Non-overlapping Template Matching Test

Reference [6] explains that the non-overlapping template matching test determines if any correlation between the given bit string and any non-periodic pattern exists. A pseudorandom bit generator could output blocks of bits that each pass the randomness test, but fails

to appear random because a repeated block template appears in the sequence in its entirety. The test searches for an m-bit pattern throughout the entire sequence using an m-bit sliding window. The test outputs a p-value that determines the randomness of the sequence. The p-value of ≥ 0.01 is considered random. Any result yielding a p-value less than 0.01 means that the sequence is non-random. For the size of the sliding window, the code is setup for $m = 1, 2, 3, \ldots, 10$ with the recommendation of $m = 9$ or 10 to achieve any meaningful results [6].

3.2.8 The Overlapping Template Matching Test

This test is very similar to the non-overlapping template matching test. However, the test looks for any irregular occurrence of a pre-specified template pattern of size m-bits in an m-bit sliding window. If the test yields a p-value less than 0.01, the sequence is considered random. The test recommends using a bit string of size $n \geq 10^6$ bits [6], which we do.

3.2.9 Maurer's "Universal Statistical" Test

This test looks for the matching patterns within the sequence and the number of bits between the patterns found. This correlates with the compression of the sequence without loss of information. The more compressible the sequence is, the less random it is. The test outputs a p-value, which determines the randomness of the sequence. If the p-value is ≥ 0.01, then the test outputs the sequence as random [6].

3.2.10 The Linear Complexity Test

According to [6], this test focuses on the length of a LFSR needed to determine the complexity of a given sequence. The linear complexity test determines the complexity of the sequence. The NIST input recommendations for this test are:

- The test requires sequences of $n \geq 10^6$ bits.
- $500 \leq M \leq 5000$, where M is the size of the block.
- The number of blocks, $N \geq 200$, where $n = M \cdot N$.

The test outputs a p-value that determines the randomness of the sequence. The p-value of ≥ 0.01 regards the sequence as random [6].

3.2.11 The Approximate Entropy Test

According to [6], this test focuses on the frequency of all possible m-bit patterns in the given bit sequence. The test compares two consecutive blocks of length m and $m + 1$ and checks for the frequency of overlapping blocks against the result of a random sequence. The same source further recommends that one choose the block size M such that $M \leq \lfloor \log n \rfloor - 2$. As with many of the other tests, this test outputs a p-value, which determines whether or not the sequence is considered to be random. For this test, if the p-value ≥ 0.01, the sequence is considered random [6].

3.2.12 The Cumulative Sums (Cusums) Test

This test looks at a portion of a sequence and determines if the cumulative sum of that sequence is relatively larger or smaller to that of a random sequence. The input is changed by converting 0's into -1's. In this context, random walk means that the test is looking at the partial sum up of the sequence and excursion means the largest deviation of the partial sums from zero. For a random sequence, the excursion of the random walk is close to zero, meaning that there are not a lot of repetitions of the same bits one after the other. The recommended input size for this test is $n \geq 100$ bits. The p-value of ≥ 0.01 as an output indicates that the binary input is random [6].

3.2.13 The Random Excursions Test

This test is similar to the cumulative-sums test. Here, the test focuses on the number of cycles in the cumulative sun random walk. According to [6], this test determines if the number of visits to a particular state within a cycle is consistent with that of a random sequence. The test consists of eight tests, each with its own p-value. Similar to previous tests, a p-value of ≥ 0.01 means that the sequence is random. The recommended input size for this test is $\geq 10^6$ bits [6].

3.2.14 The Random Excursions Variant Test

According to [6], the test checks for the cumulative-sum random walk and compares the number of times a particular state occurs. It detects the number of times a deviation occurs when compared to a random sequence. This test is comprised of a series of 18 tests each

with an associated p-value. Similar to other tests, a p-value of ≥ 0.01 means that the sequence is random. The recommended input size for the test is $\geq 10^6$ bits [6].

CHAPTER 4:
Test Results

This chapter focuses on the comparison between the original and the modified BBS. We have eight different types of samples, of which each differ in the number of bits which is used. The samples are divided into sequences that extract the last bit, last 3 bits, last 4 bits, last 5 bits, last 10 bits, last 50 bits, last 100 bits, and all bits, respectively. For each sample type, we generate 200 sequences for both versions of BBS. We want to see how the generated sequences perform against the NIST tests. We do so using three comparisons:

1. In the first comparison, we compare the pass rates for both the modified and the original version of BBS based on specific NIST tests.
2. The second comparison examines performance based on overall pass rate.
3. The third comparison examines the modified and the original version of BBS with respect to the time it takes to generate the sequence.

As discussed in Chapter 3, Vazirani and Vazirani proved that we can extract up to $\log\log N$ bits in each iteration of BBS, where N is the size of the modulus, and still maintain security [11]. The authors also proved that extracting 2 bits per iteration of BBS, generates a sequence that is secure [11]. The size of the modulus N for the original and the modified BBS we use are 2^{512} and 2^{768}, respectively. For the original BBS, this gives us $\log\log N = \log\log 2^{512} = 9$ bits. Similarly, for the modified BBS, this gives us $\log\log N = \log\log 2^{768} = 9.58$ bits, and we use the floor function upper bound of 9 bits.

4.1 The Last Bit

As one might expect, the last bit sequences only use the last bit. Each sequence for this test is of size one million bits. For both versions of BBS, the size of the Blum-primes that we generate are 256 bits long.

4.1.1 Performance Based on Type of Test

To display our results, we plot the data as a bar graph. Figure 4.1 shows the percentage of sequences which pass each test for both versions of BBS. The data collected shows that

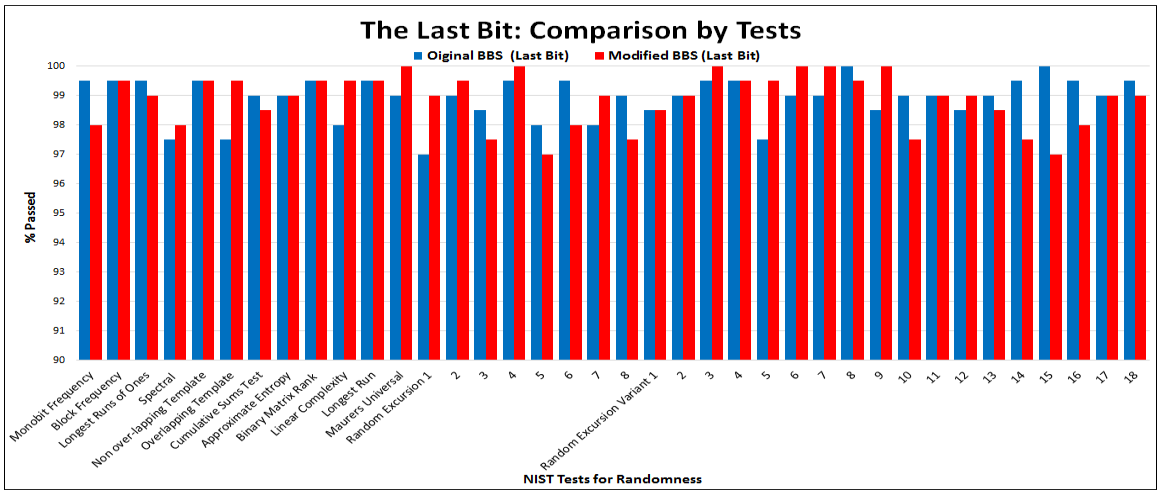
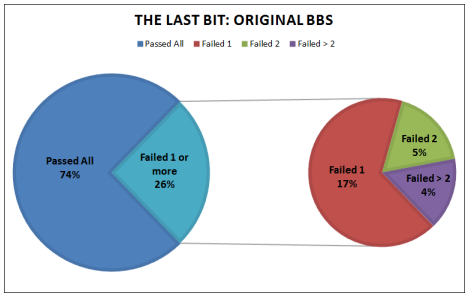


Figure 4.1: The Last Bit: Side-by-Side Comparison by Tests

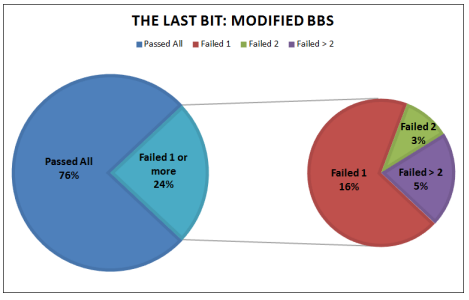
there is little variation in the percentage of sequences that passed each test. For all 14 tests, we see that both version of BBS have a pass rate of more than 97 percent. According to the data, it is difficult to distinguish any performance difference between the two versions of BBS.

4.1.2 Overall Performance

For this comparison, we focus on how the individual sequences performed against all of the NIST tests. We want to see what percentage of the generated pseudo-random sequence passed all of the tests. Among the failed tests, we also want to examine how many sequences failed one, two, or more than two tests. The results are shown in Figure 4.2.



(a) Original BBS



(b) Modified BBS

Figure 4.2: The Last Bit: Overall Performance by Sequences

The data for the original BBS shows that 74 percent of generated sequences passed all 14 tests. Among the failed sequences, 17 percent failed one test, 5 percent failed 2 tests, and 4 percent failed more than 2 tests. The data for the modified BBS indicates 76 percent of generated sequences passed all 14 tests. Among the failed sequences, 16 percent failed one test, 3 percent failed 2 tests, and 5 percent failed more than 2 tests. This indicates that the performances of both the original and the modified BBS are similar. We could not make a clear distinction on the superiority of one sequence type over the other.

4.2 Last 3 Bits

We generated the sequences for this sample type by taking the last 3 bits in each iteration. Here, the length of the sequence needs to be such that dividing by the number of bits we wish to take produces an integer. We chose the length to be 1,000,002 so 1,000,002 / 3 = 333,334 iterations, where the last 3 bits are extracted during each iteration of the program.

4.2.1 Performance Based on Type of Test

In this side by side comparison, the data collected for the original BBS suggests that the pass rate is more than 96 percent or higher. Similarly, the modified version has a pass rate of 95 percent or higher. Figure 4.3 shows the side by side comparison of this sample type for each of the NIST tests.

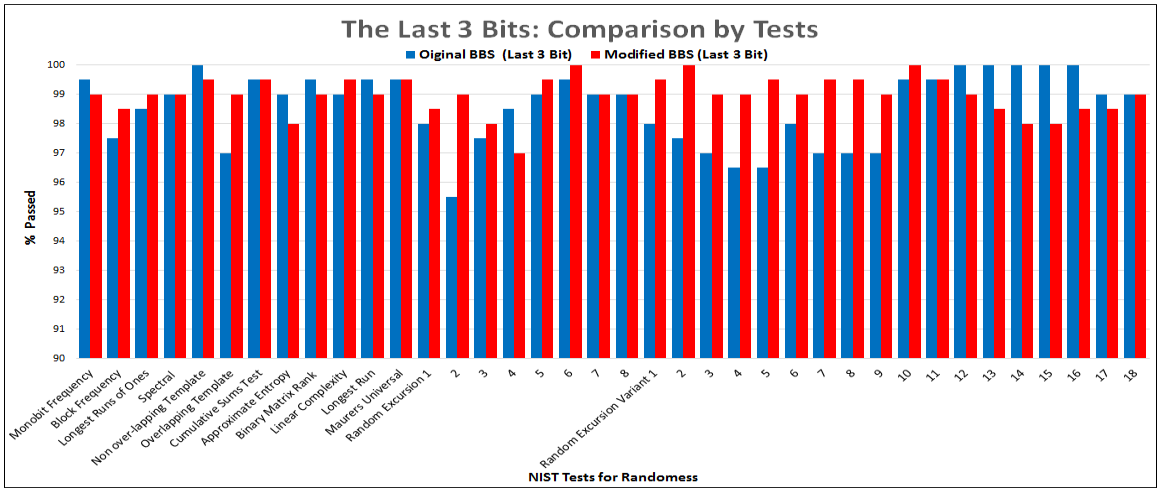
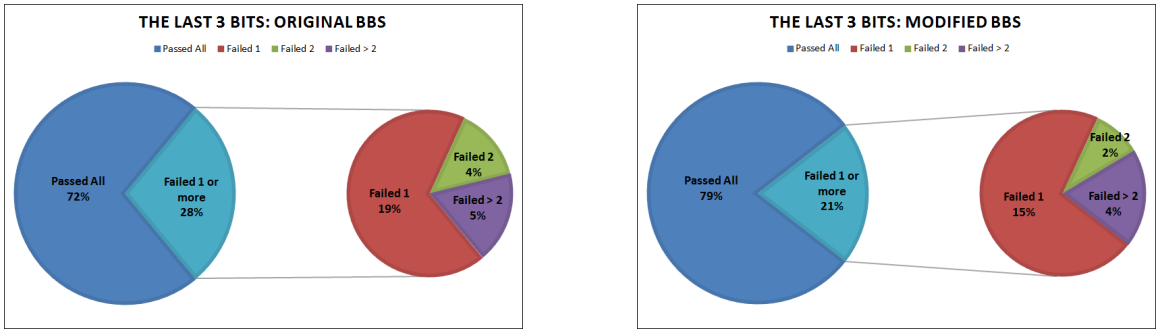


Figure 4.3: Last 3 Bits: Side-by-Side Comparison by Tests

4.2.2 Overall Performance

In this comparison, the data suggests that the modified BBS performed slightly better than the original version. Looking at the data, we see that the pass rate for the modified version is at 79 percent as opposed to the 72 percent for the original version. When considering the failure rate for the original BBS, we see that 19 percent of sequences failed one test, 4 percent failed two tests, and 5 percent failed more than 2 tests. For the modified version, the failure rate was 15 percent, 2 percent and 4 percent, respectively.



(a) Original BBS (b) Modified BBS

Figure 4.4: Last 3 Bits: Overall Performance by Sequences

4.3 Last 4 Bits

4.3.1 Performance Based on Type of Test

The data for the original BBS suggests that the pass rate is 97.5 percent or higher for all NIST tests. Similarly, the modified version has the pass rate of 96 percent or higher for all NIST tests, indicating a similar performance to that of the original BBS. The results are shown in Figure 4.5.

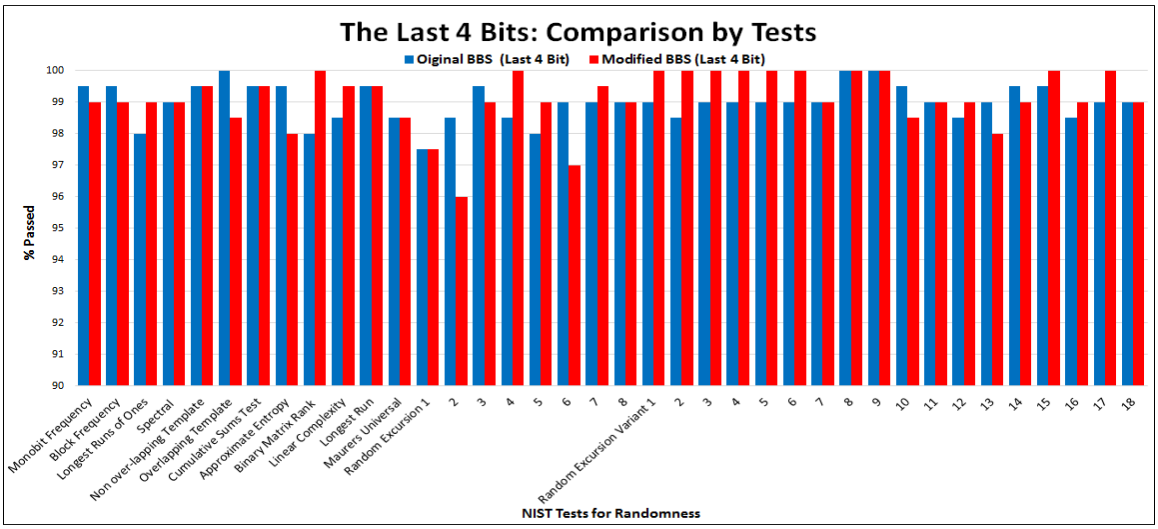
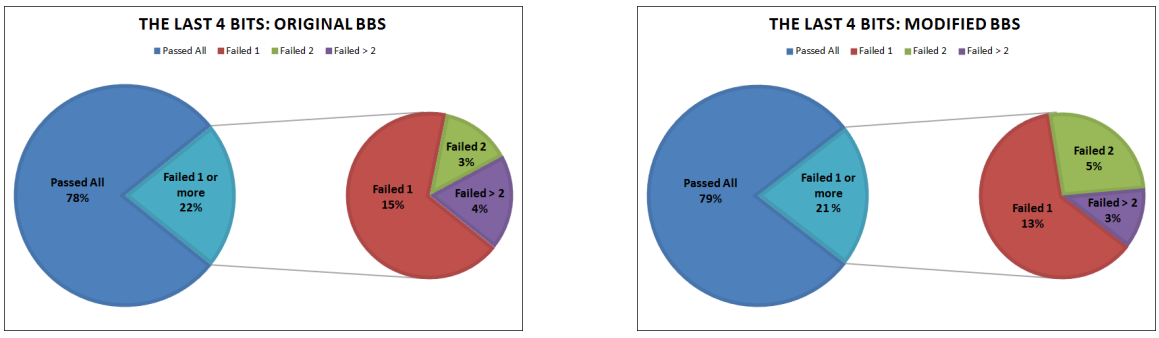


Figure 4.5: Last 4 Bits: Side-by-Side Comparison by Tests

4.3.2 Overall Performance

The data shows that there is not much variation on the overall performance against the NIST standard by both version of BBS. The original version has an overall pass rate of 78 percent compared to 79 percent pass rate for the modified version. Among the failure rates, both versions have similar results. The results for both versions of BBS are shown in Figure 4.6..



(a) Original BBS (b) Modified BBS

Figure 4.6: Last 4 Bits: Overall Performance by Sequences

4.4 Last 5 Bits

4.4.1 Performance Based on Type of Test

The data for the original BBS shows that the pass rate for sequences is 97.5 percent or higher for each NIST test. The modified BBS performed similar to the original with a pass rate of 97 percent or higher for each NIST tests. Figure 4.7 shows the result for each tests.

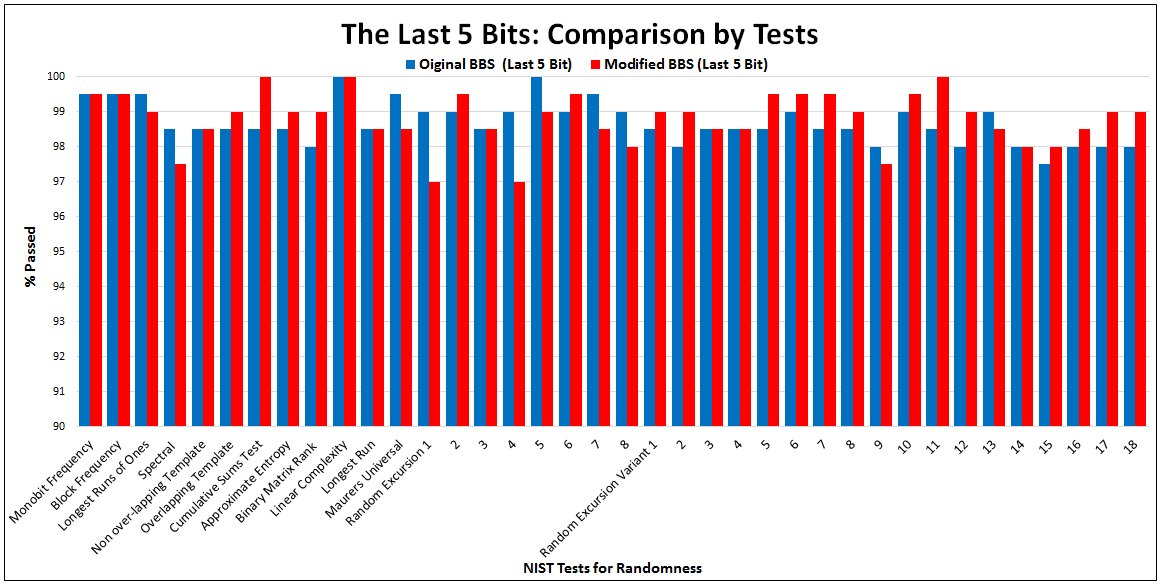
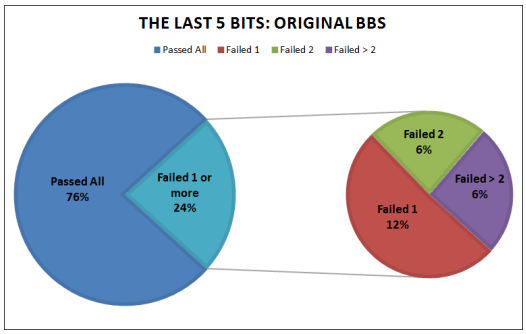


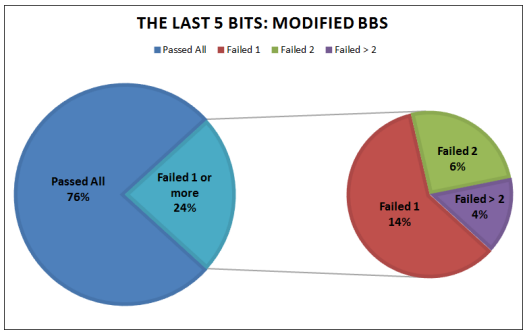
Figure 4.7: Last 5 Bits: Side-by-Side Comparison by Tests

4.4.2 Overall Performance:

For this sample type, both the original and the modified BBS performed with equal pass rates of 76 percent. Looking at the failure rate for the original BBS, we see that 12 percent of sequences failed one test, 6 percent failed two tests, and 6 percent failed more than two tests. Similarly, the failure rates of the modified BBS are 14 percent, 6 percent, and 4 percent, respectively. So far, both versions have performed in a similar manner in the pseudo-randomness test. Figure 4.8 shows the result for both versions.



(a) Original BBS



(b) Modified BBS

Figure 4.8: Last 5 Bits: Overall Performance by Sequences

4.5 Last 10 Bits

4.5.1 Performance Based on Type of Test

The data collected shows that both the original and the modified versions of BBS have similar performance. Both versions have pass rates of 97 percent or higher, and 96.5 percent or higher for all 14 NIST tests, respectively.

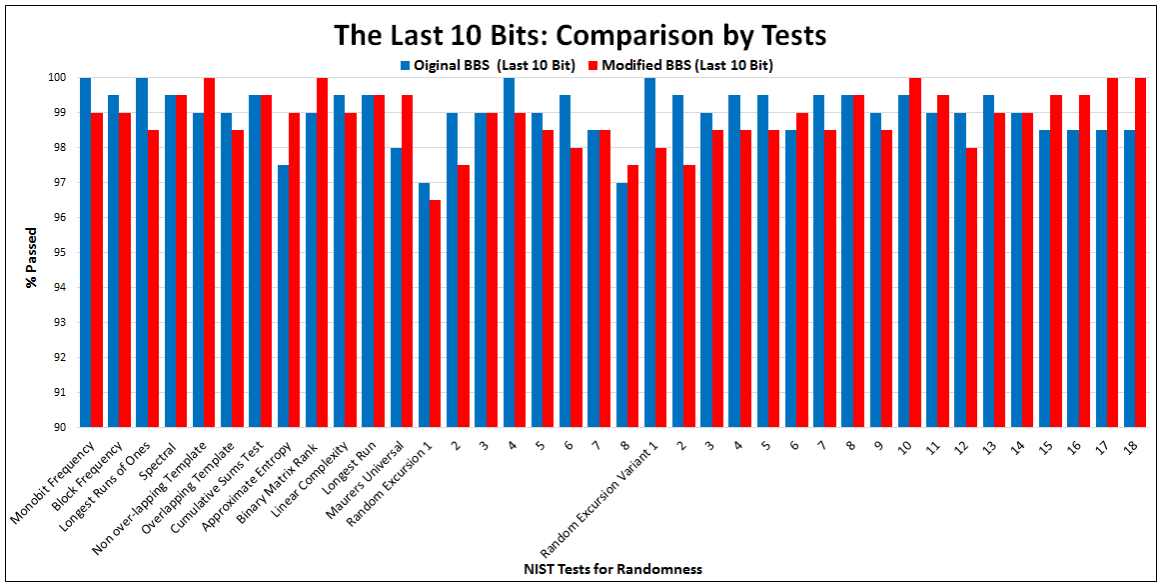
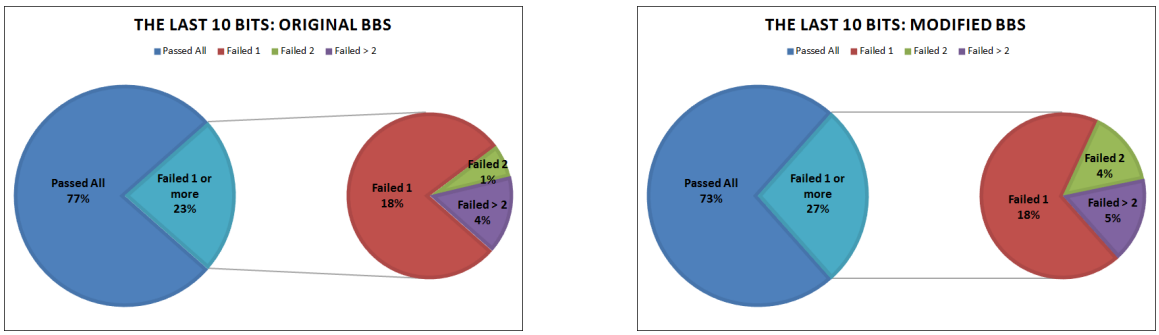


Figure 4.9: Last 10 Bits: Side-by-Side Comparison by Tests

4.5.2 Overall Performance

Here, the data shows that the original BBS performed slightly better than the modified version. The original BBS has a pass rate of 77 percent compared to 73 percent for the modified version. Figure 4.10 shows the result.



(a) Original BBS (b) Modified BBS

Figure 4.10: Last 10 Bits: Overall Performance by Sequences

4.6 Last 50 Bits

In order to take the last 50 bits and generate a sequence of length greater than one million bits, we had our program generate the sequences of length 2 million bits. We did this to account for the cases where the quadratic residue modulo n was smaller than 50 bits in size. As the program ($mainbbs.py$) pre-calculates the number of iteration based on the the length of the sequence to be generated and the number of bits taken per iteration, this approach ensures that the length of the output is greater than or equal to one million bits.

4.6.1 Performance Based on Type of Test:

Similar to other results, the data for this sample type do not show any noticeable variation. Both the original and the modified BBS have pass rates of 97 percent or higher, and 97.5 percent or higher, respectively. Figure 4.11 shows the results.

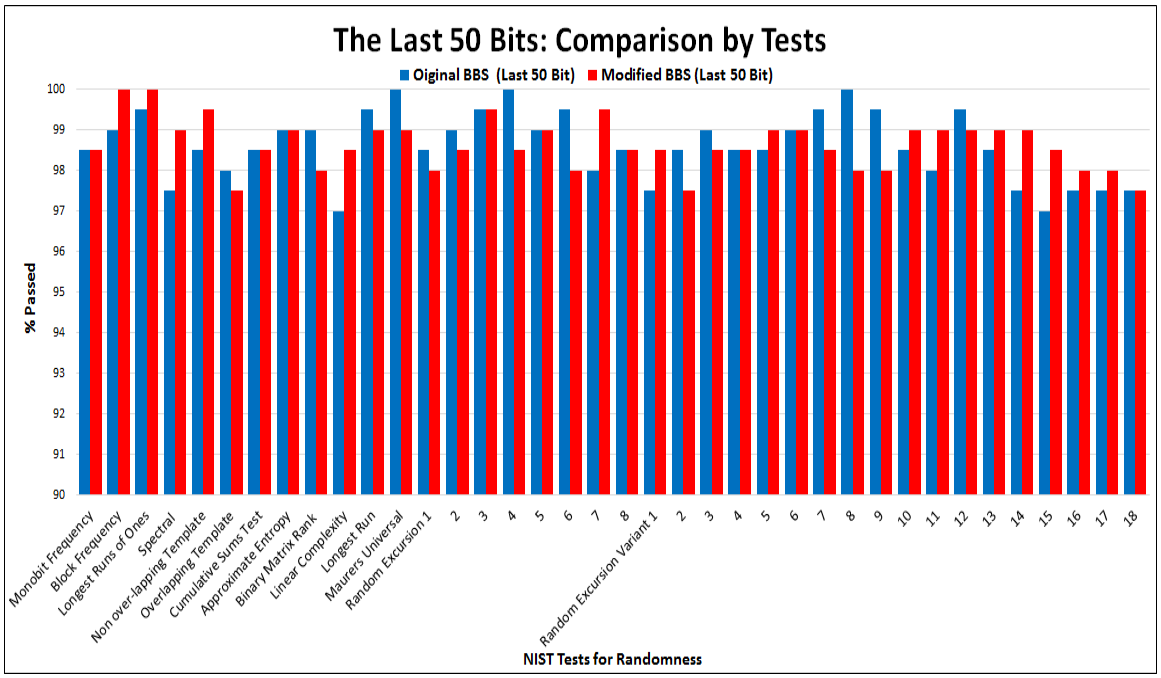
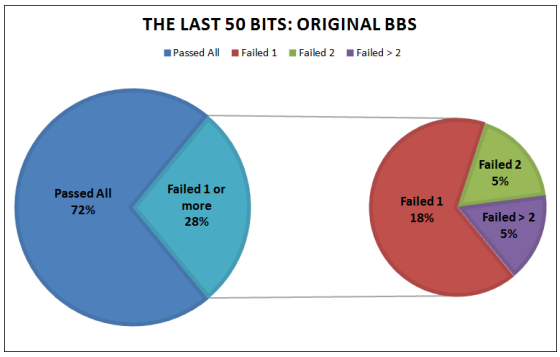


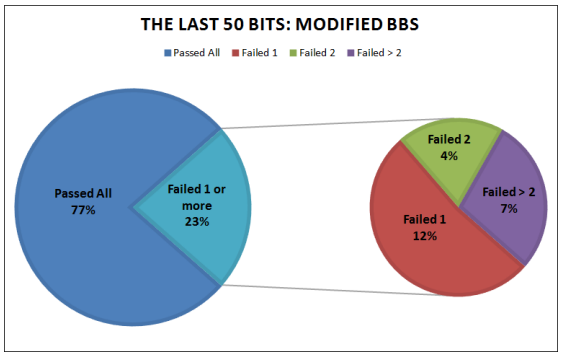
Figure 4.11: Last 50 Bits: Side-by-Side Comparison by Tests

4.6.2 Overall Performance

Here, the modified version performed slightly better than the original BBS. The modified version performed with a 77 percent pass rate compared to a 72 percent pass rate for the original version. The results are shown in Figure 4.12.



(a) Original BBS



(b) Modified BBS

Figure 4.12: Last 50 Bits: Overall Performance by Sequences

4.7 Last 100 Bits

4.7.1 Performance Based on Type of Test

The data shows that the original and the modified BBS each have pass rates of 97 or 97.5 percent or higher. This result is very similar to the previous sample types. Figure 4.13 shows the results.

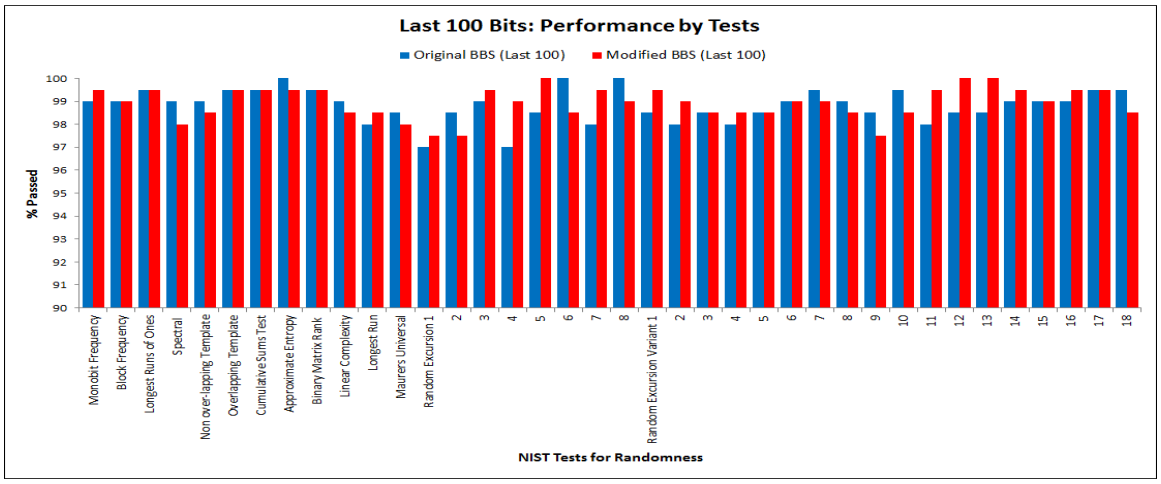
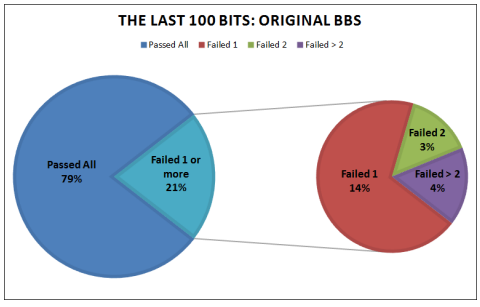


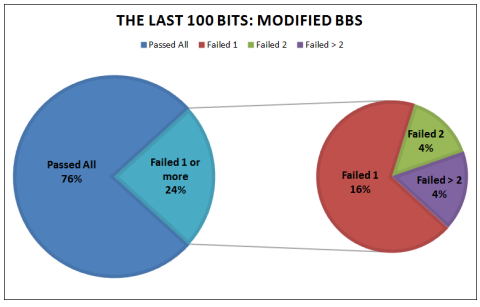
Figure 4.13: Last 100 Bits: Side-by-Side Comparison by Tests

4.7.2 Overall Performance

Here, the original version performed with 79 percent pass rate compared to a 76 percent pass rate of the modified version. Figure 4.14 shows the results.



(a) Original BBS



(b) Modified BBS

Figure 4.14: Last 100 Bits: Overall Performance by Sequences

4.8 All Bits

For this category, the algorithm generates sequences by extracting all of the bits in each iteration. The generated sequences are roughly 1.1 million bits in length. The comparison focuses on the effects on the pseudorandomness properties of sequences when tested against the NIST tests.

4.8.1 Performance Based on Type of Test

The data suggests that both versions performed in a similarly. Here, the Monobit frequency test and Cumulative sums test yield low pass rate between 90.5 and 92 percent for both versions. All other tests have pass rate of 96.5 percent or higher for both versions. Figure 4.15 shows the results.

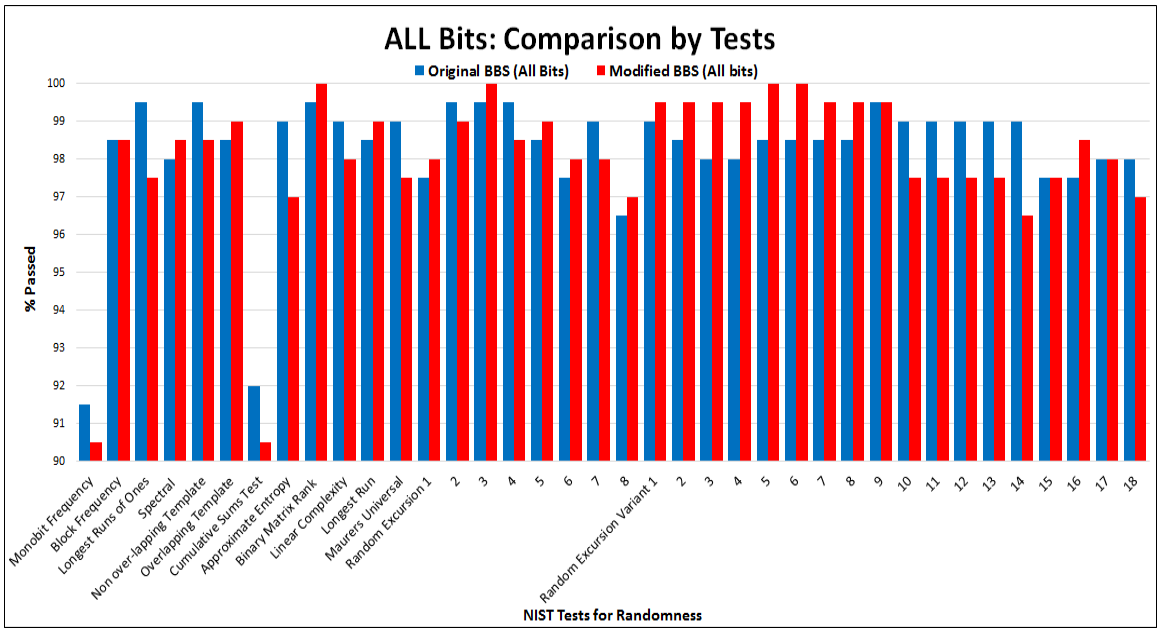
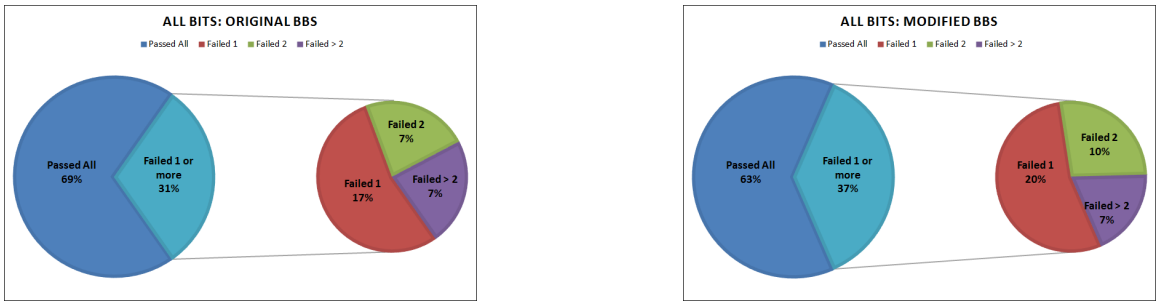


Figure 4.15: All Bits: Side-by-Side Comparison by Tests

4.8.2 Overall Performance

Here, the original BBS performed slightly better than the modified BBS. The original version performed with a 69 percent pass rate compared to a 63 percent pass rate of the modified version. Figure 4.16 shows the results.

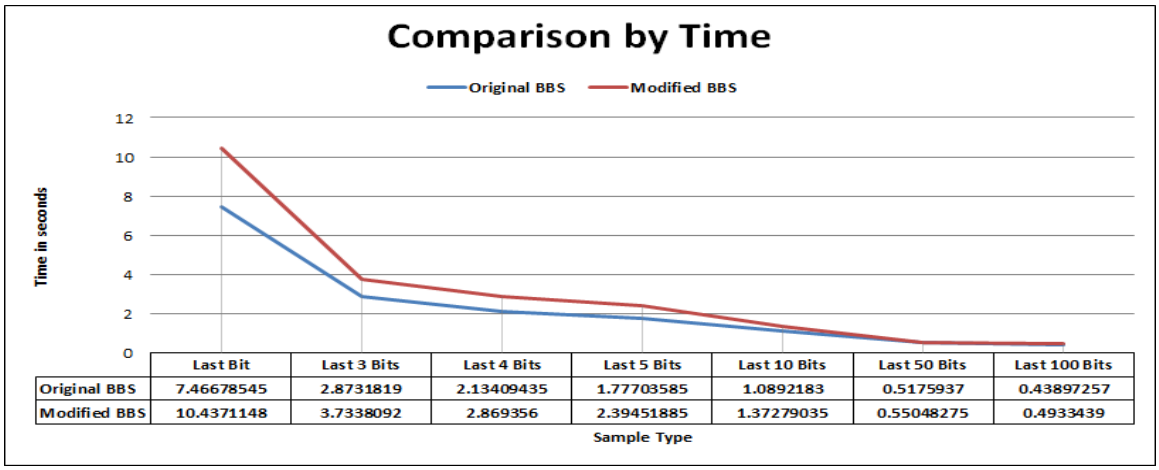


(a) Original BBS (b) Modified BBS

Figure 4.16: All Bits: Overall Performance by Sequences

4.9 Comparison Based on Time Required to Generate the Sequences

The Blum-Blum-Shub pseudorandom bit generator is (probabilistically) provably secure but slow compared to other PRBGs. For some low level application, it is certainly too slow, however, it may be suitable for some applications requiring a higher degree of security [10]. In this section, we compute the time required to generate 200 sequences, each of length approximately one million bits for the seven sample types discussed earlier. We then examine the average time required to generate one sequence. Figure 4.17 shows the results.



	Last Bit	Last 3 Bits	Last 4 Bits	Last 5 Bits	Last 10 Bits	Last 50 Bits	Last 100 Bits
Original BBS	7.46678545	2.8731819	2.13409435	1.77703585	1.0892183	0.5175937	0.43897257
Modified BBS	10.4371148	3.7338092	2.869356	2.39451885	1.37279035	0.55048275	0.4933439

Figure 4.17: Average Time to Generate a Million-Bit Sequence

As previously mentioned, this test was performed using an Apple Macbook with the following specifications:

- Operating System: OS X El Capitan
- Version: 10.11.4
- Type: Macbook Air (13 inch, Early 2015)
- Processor: 1.6 GHz Intel Core i5
- Memory: 4 GB 1600 MHz DDR3
- Graphics: Intel HD Graphics 6000 1536 MB

When looking at the last bit, the data shows that the original BBS takes on average 7.46 seconds to generate a one million bit sequence. As expected, the time significantly decreases as we take more bits per iteration. For the original BBS, taking the last 3 bits reduces the time by approximately 60 percent. For the last 4, last 5, last 50 and last 100 bits, the time drops by approximately 70, 76, 85, 93, and 94 percent respectively when compared to the time required when only taking the last bit. Similarly, when only taking the last bit, the modified version takes an average of 10.43 seconds to generate a million-bit sequence. The time drops by approximately 65 percent when we take the last 3 bits per iteration. The time required when taking the last 4, last 5, last 10, last 50 and last 100 bits are 73 percent, 77 percent, 87 percent, 94 percent, and 95 percent, respectively.

When looking at the result for both versions, we see that the time required to generate a million bit sequence decreases significantly as the bits taken per iteration is increased. Also, the data shows that the time required to generate a million-bit sequence is similar when the bits taken per iteration is 10 or higher.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5:
Conclusion and Future Work

5.1 Conclusion

This thesis compared the original and the modified BBS. For this comparison, we generated sequences roughly one million bits in length and tested these sequences for pseudo-randomness properties using a suite of tests from NIST. In order to compare the two versions, we expanded the number of bits that could be extracted per iteration of the algorithm. As discussed in Chapter 2, Vazirani and Vazirani showed that the number of least significant bits that could be extracted per iteration is up to $\log \log N$ [11]. During our research we extracted far more bits than the $\log \log N$ limit.

The test results that we analyzed were based on the NIST testing standards. Our comparisons between both the original and modified versions of BBS was carried out using three metrics: the type of test, the overall performance of each sequence when measured against the NIST standards, and the time taken to generate the sequences in each category.

When we looked at the performance based on the type of test, the data suggests that both versions of BBS performed in a similar manner. In each category, the passing rate we observed was above 95 percent for both versions. Similarly, the data suggests that the overall performance of the sequences in each category was similar as well. In few cases, one version outperformed the other by a small margin. For the last 3 bits test, the modified BBS performed slightly better with a pass rate of 79 percent compared to a pass rate of 72 percent for the original BBS. For the last 10 bits category, the overall performance of the original BBS was better with a 77 percent pass rate as compared to a 73 percent pass rate for the modified BBS. For the last 50 bit test, the modified BBS performed better with 77 percent overall pass rate compared to the 72 percent pass rate of the original BBS. For the last 100 bits test category, the original BBS performed marginally better than the modified version with a 79 percent versus 76 percent overall pass. It is worth noting that the overall pass rate for both BBS versions stayed between 70 and 80 percent. The data obtained from our research suggests that there was very little performance difference between the original

and modified versions of BBS. The result of the statistical tests we performed also served to validate the good pseudo-randomness properties of the original BBS.

5.2 Future Work

In Chapter 2, we briefly introduced cryptosystems based on the original BBS and the modified BBS. It may be useful to further expand upon these ideas. In our research, we compared the original and modified versions of BBS using statistical tests. The scope and time allotted for our research unfortunately did not afford us an opportunity to pursue number theoretic proofs for the security of the modified BBS. We therefore recommend that future research on this topic include a formal proof for the security of the modified BBS. We also think it would be valuable to explore research on the cryptosystem we proposed when one of the three prime is not a Blum-prime. The data in Chapter 4 suggests that the time required to generate a million bit long sequence dropped significantly when more bits were extracted per iteration while the generated sequence still maintained its desirable pseudo-randomness properties. For future work, we suggest using different size Blum-primes to narrow down the practical limit for bit extraction per iteration beyond $\log \log N$, when compared to the size of modulus.

APPENDIX: BBS Test Programs Written in Python

A.1 The Main Interface: *mainbbs.py*

This is the working interface to generate, test and record the test results as an Excel file. Devendra Manandhar, reporting analyst 3 at Providence Regional and Medical Center, WA provided invaluable time and guidance to help us write the main interface [12]. The code is as follows:

```
import csv
import random
import time
import numpy as np


randsequence = input("How many sequences?")
DesiredBits = input("How long do you want the bits sequence to be?")
divisor= input("How many of last n bits do you want to take
                to make bit sequence?")
IterationsNeeded = DesiredBits/divisor #this gives you the desired
                   bits eg. if you want 1 million bits sequence,
                   and wanna take last 4 bits, algorithm
                   will divide the iterations into 250000

file1 = open( "Name of File for the result.csv", 'w')

file1.write("Monobit,Blockfreq, Runstest, Spectraltest,
            Nonoverlapping Template, Overlapping Template,
            Cumulative Sums Test,Approximate Entropy, binarymatrixranktest,
            Linear Complexity, Longest Run, Maurers Universal,
            Random Excursion1,2,3,4,5,6,7,8,
            Random Excursion Variant1,2,3,4,5,6,7,8,9,10,
            11,12,13,14,15,16,17,18,p,p1,q,q1,r,r1,xi, \n")
```

```
from PrimeGen import bigppr, pptest, algP

from randtest import monobitfrequencytest, blockfrequencytest, runstest,
                     spectraltest, nonoverlappingtemplatematchingtest,
                     overlappingtemplatematchingtest,cumultativesumstest,
                     aproximateentropytest, binarymatrixranktest,
                     linearcomplexitytest,longestrunones10000,
                     maurersuniversalstatistictest, randomexcursionstest,
                     randomexcursionsvarianttest



def rng():
    global xi, yi

    xi = (xi * xi) % n

    bin (xi)
    yi= bin(xi)[2:] #it eliminates the first two characters 0b...
    zi = str (yi)
    ai = str (zi[-divisor:])
    return ai

for i in range(randsequence):
    p = bigppr(256)
    q = bigppr(256)
    r = bigppr (256)

    n = p*q*r
    #n = p*q

    xi = bigppr (300)

    output = ''
```

```
    result = ''

    for i in range(IterationsNeeded):
        output += str(rng())

    p1 = p%4 #Double check if Blum-prime
    q1 = q%4 #Double check if Blum-prime
    r1 = r%4 #Double check if Blum-prime

    file1.write(str(monobitfrequencytest(output)) + ',' +
                str(blockfrequencytest(output)) + ',' +
                str(runstest(output)) + ',' + str(spectraltest(output))
                + ',' + str(nonoverlappingtemplatematchingtest(output))
                + ',' + str(overlappingtemplatematchingtest(output))
                + ',' + str(cumultativesumstest(output)) + ',' +
                str(aproximateentropytest(output)) + ',' +
                str(binarymatrixranktest(output)) + ',' +
                str(linearcomplexitytest(output))  +  ',' +
                str(longestrunones10000(output)) + ',' +
                str(maurersuniversalstatistictest(output)) + ',' +
                str(randomexcursionstest(output)) + ',' +
                str(randomexcursionsvarianttest(output))
                 + ',' + str(p)+ ','+ str(p1)+ ',' + str(q) + ','+ str(q1)
                 + ',' + str(r)  + ','+ str(r1) + ',' + str(xi) + '\n')

file1.close()

#toc= time.clock()
#print (toc - tic)
print p
print q
print xi
#print output
#print toc
```

```
import os
os.system('say "Your test is done" ')
```

A.2 Blum-Prime Generator: *primes.py*

The code below generates the Blum-prime and was downloaded for our use [13]. The code is as follows:

```
import random

def bigppr(bits=256):
    """
    Randomly generate a probable prime with a given
    number of hex digits
    """

    candidate = random.getrandbits(bits) | 1  # Ensure odd

    prob = 0
    while 1:
        prob=pptest(candidate)
        if prob>0:
            return candidate
        candidate += 2

def pptest(n):
    """
    Simple implementation of Miller-Rabin test for
    determining probable primehood.
    """

    if n<=1:
        return 0

    # if any of the primes is a factor, we're done
```

```
    bases  = [random.randrange(2,50000) for x in xrange(90)]


    for b in bases:
        if n%b==0:
            return 0

    tests,s  = 0L,0
    m        = n-1

    # turning (n-1) into (2**s) * m

    while not m&1:  # while m is even
        m >>= 1
        s += 1

    for b in bases:
        tests += 1
        isprob = algP(m,s,b,n)
        if not isprob:
            break

    if isprob:
        return (1-(1./(4**tests)))

    return 0

def algP(m,s,b,n):
    """
    based on Algorithm P in Donald Knuth's 'Art of
    Computer Programming' v.2 pg. 395
    """
    y = pow(b,m,n)

    for j in xrange(s):
```

```
        if (y==1 and j==0) or (y==n-1):
            return 1
        y = pow(y,2,n)

        return 0
```

A.3 NIST Tests: *randtest.py*

The code below is downloaded to test the generated sequences against the suite of NIST tests [14]. The code is as follows:

```
#!/usr/bin/env python

import numpy as np
import scipy.special as spc
import scipy.fftpack as sff
import scipy.stats as sst

def sumi(x): return 2 * x - 1
def su(x, y): return x + y
def sus(x): return (x - 0.5) ** 2
def sq(x): return int(x) ** 2
def logo(x): return x * np.log(x)

def pr(u, x):
    if u == 0:
        out=1.0 * np.exp(-x)
    else:
        out=1.0 * x * np.exp(2*-x) * (2**-u) * spc.hyp1f1(u + 1, 2, x)
    return out

def stringpart(binin, num):
    blocks = [binin[xs * num:num + xs * num:] for xs in xrange(len(binin)
             / num)]
    return blocks
```

```
def randgen(num):
    '''Spits out a stream of random numbers like '1001001'
    with the length num'''

    rn = open('/dev/urandom', 'r')
    random_chars = rn.read(num / 2)
    stream = ''
    for char in random_chars:
        c = ord(char)
        for i in range(0, 2):
            stream += str(c >> i & 1)
    return stream

def monobitfrequencytest(binin):

    ss = [int(el) for el in binin]
    sc = map(sumi, ss)
    sn = reduce(su, sc)
    sobs = np.abs(sn) / np.sqrt(len(binin))
    pval = spc.erfc(sobs / np.sqrt(2))
    return pval

def blockfrequencytest(binin, nu=128):

    ss = [int(el) for el in binin]
    tt = [1.0 * sum(ss[xs * nu:nu + xs * nu:]) / nu for xs in
         xrange(len(ss) / nu)]
    uu = map(sus, tt)
    chisqr = 4 * nu * reduce(su, uu)
    pval = spc.gammaincc(len(tt) / 2.0, chisqr / 2.0)
    return pval

def runstest(binin):
```

```
    ss = [int(el) for el in binin]
    n = len(binin)
    pi = 1.0 * reduce(su, ss) / n
    vobs = len(binin.replace('0', ' ').split()) +
           len(binin.replace('1' , ' ').split())
    pval = spc.erfc(abs(vobs-2*n*pi*(1-pi)) / (2 * pi * (1 - pi)
           * np.sqrt(2*n)))
    return pval

def longestrunones8(binin):

    m = 8
    k = 3
    pik = [0.2148, 0.3672, 0.2305, 0.1875]
    blocks = [binin[xs*m:m+xs*m:] for xs in xrange(len(binin) / m)]
    n = len(blocks)
    counts1 = [xs+'01' for xs in blocks] # append the string 01 to
    guarantee the length of 1
    counts = [xs.replace('0',' ').split() for xs in counts1] # split into
    all parts
    counts2 = [map(len, xx) for xx in counts]
    counts4 = [(4 if xx > 4 else xx) for xx in map(max,counts2)]
    freqs = [counts4.count(spi) for spi in [1, 2, 3, 4]]
    chisqr1 = [(freqs[xx]-n*pik[xx])**2/(n*pik[xx]) for xx in xrange(4)]
    chisqr = reduce(su, chisqr1)
    pval = spc.gammaincc(k / 2.0, chisqr / 2.0)
    return pval

def longestrunones128(binin):  # not well tested yet
    if len(binin) > 128:
        m = 128
        k = 5
        n = len(binin)
        pik = [ 0.1174, 0.2430, 0.2493, 0.1752, 0.1027, 0.1124 ]
        blocks = [binin[xs * m:m + xs * m:] for xs in
```

```
                xrange(len(binin) / m)]
        n = len(blocks)
        counts = [xs.replace('0', ' ').split() for xs in blocks]
        counts2 = [map(len, xx) for xx in counts]
        counts3 = [(1 if xx < 1 else xx) for xx in map(max, counts2)]
        counts4 = [(4 if xx > 4 else xx) for xx in counts3]
        chisqr1 = [(counts4[xx] - n * pik[xx]) ** 2 / (n * pik[xx])
        for xx in xrange(len(counts4))]
        chisqr = reduce(su, chisqr1)
        pval = spc.gammaincc(k / 2.0, chisqr / 2.0)
    else:
        print 'longestrunones128 failed, too few bits:', len(binin)
        pval = 0
    return pval

def longestrunones10000(binin):  # not well tested yet

    if len(binin) > 128:
        m = 10000
        k = 6
        pik = [0.0882, 0.2092, 0.2483, 0.1933, 0.1208, 0.0675, 0.0727]
        blocks = [binin[xs * m:m + xs * m:] for xs in xrange(len(binin)
                / m)]
        n = len(blocks)
        counts = [xs.replace('0', ' ').split() for xs in blocks]
        counts2 = [map(len, xx) for xx in counts]
        counts3 = [(10 if xx < 10 else xx) for xx in map(max, counts2)]
        counts4 = [(16 if xx > 16 else xx) for xx in counts3]
        freqs = [counts4.count(spi) for spi in [10,11,12,13,14,15,16]]
        chisqr1 = [(freqs[xx] - n * pik[xx]) ** 2 / (n * pik[xx]) for xx in
                  xrange(len(freqs))]
        chisqr = reduce(su, chisqr1)
        pval = spc.gammaincc(k / 2.0, chisqr / 2.0)
    else:
        print 'longestrunones10000 failed, too few bits:', len(binin)
```

```
        pval = 0
    return pval

# test 2.06
def spectraltest(binin):

    n = len(binin)
    ss = [int(el) for el in binin]
    sc = map(sumi, ss)
    ft = sff.fft(sc)
    af = abs(ft)[1:n/2+1:]
    t = np.sqrt(np.log(1/0.05)*n)
    n0 = 0.95*n/2
    n1 = len(np.where(af<t)[0])
    d = (n1 - n0)/np.sqrt(n*0.95*0.05/4)
    pval = spc.erfc(abs(d)/np.sqrt(2))
    return pval

def nonoverlappingtemplatematchingtest(binin, mat="000000001", num=8):

    n = len(binin)
    m = len(mat)
    M = n/num
    blocks = [binin[xs*M:M+xs*M:] for xs in xrange(n/M)]
    counts = [xx.count(mat) for xx in blocks]
    avg = 1.0 * (M-m+1)/2 ** m
    var = M*(2**-m -(2*m-1)*2**(-2*m))
    chisqr = reduce(su, [(xs - avg) ** 2 for xs in counts]) / var
    pval = spc.gammaincc(1.0 * len(blocks) / 2, chisqr / 2)
    return pval

def occurances(string, sub):
    count=start=0
    while True:
        start=string.find(sub,start)+1
```

```
        if start>0:
            count+=1
        else:
            return count

def overlappingtemplatematchingtest(binin,mat="111111111",num=1032,numi=5):

    n = len(binin)
    bign = int(n / num)
    m = len(mat)
    lamda = 1.0 * (num - m + 1) / 2 ** m
    eta = 0.5 * lamda
    pi = [pr(i, eta) for i in xrange(numi)]
    pi.append(1 - reduce(su, pi))
    v = [0 for x in xrange(numi + 1)]
    blocks = stringpart(binin, num)
    blocklen = len(blocks[0])
    counts = [occurances(i,mat) for i in blocks]
    counts2 = [(numi if xx > numi else xx) for xx in counts]
    for i in counts2: v[i] = v[i] + 1
    chisqr = reduce(su, [(v[i]-bign*pi[i])** 2 / (bign*pi[i])
             for i in xrange(numi + 1)])
    pval = spc.gammaincc(0.5*numi, 0.5*chisqr)
    return pval


def maurersuniversalstatistictest(binin,l=7,q=1280):

    ru = [
        [0.7326495, 0.690],
        [1.5374383, 1.338],
        [2.4016068, 1.901],
        [3.3112247, 2.358],
        [4.2534266, 2.705],
        [5.2177052, 2.954],
```

```
            [6.1962507, 3.125],
            [7.1836656, 3.238],
            [8.1764248, 3.311],
            [9.1723243, 3.356],
            [10.170032, 3.384],
            [11.168765, 3.401],
            [12.168070, 3.410],
            [13.167693, 3.416],
            [14.167488, 3.419],
            [15.167379, 3.421],
            ]
        blocks = [int(li, 2) + 1 for li in stringpart(binin, l)]
        k = len(blocks) - q
        states = [0 for x in xrange(2**l)]
        for x in xrange(q):
            states[blocks[x]-1]=x+1
        sumi=0.0
        for x in xrange(q,len(blocks)):
            sumi+=np.log2((x+1)-states[blocks[x]-1])
            states[blocks[x]-1] = x+1
        fn = sumi / k
        c=0.7-(0.8/l)+(4+(32.0/l))*((k**(-3.0/l))/15)
        sigma=c*np.sqrt((ru[l-1][1])/k)
        pval = spc.erfc(abs(fn-ru[l-1][0]) / (np.sqrt(2)*sigma))
        return pval

    def lempelzivcompressiontest1(binin):

        i = 1
        j = 0
        n = len(binin)
        mu = 69586.25
        sigma = 70.448718
        words = []
        while (i+j)<=n:
```

```
        tmp=binin[i:i+j:]
        if words.count(tmp)>0:
            j+=1
        else:
            words.append(tmp)
            i+=j+1
            j=0
    wobs = len(words)
    pval = 0.5*spc.erfc((mu-wobs)/np.sqrt(2.0*sigma))
    return pval

def lempelzivcompressiontest(binin):

    i = 1
    j = 0
    n = len(binin)
    mu = 69586.25
    sigma = 70.448718
    words = []
    while (i+j)<=n:
        tmp=binin[i:i+j:]
        if words.count(tmp)>0:
            j+=1
        else:
            words.append(tmp)
            i+=j+1
            j=0
    wobs = len(words)
    pval = 0.5*spc.erfc((mu-wobs)/np.sqrt(2.0*sigma))
    return pval


# test 2.11
def serialtest(binin, m=4):
```

```
    n = len(binin)
    hbin=binin+binin[0:m-1:]
    f1a = [hbin[xs:m+xs:] for xs in xrange(n)]
    oo=set(f1a)
    f1 = [f1a.count(xs)**2 for xs in oo]
    f1 = map(f1a.count,oo)
    cou =f1a.count
    f2a = [hbin[xs:m-1+xs:] for xs in xrange(n)]
    f2 = [f2a.count(xs)**2 for xs in set(f2a)]
    f3a = [hbin[xs:m-2+xs:] for xs in xrange(n)]
    f3 = [f3a.count(xs)**2 for xs in set(f3a)]
    psim1 = 0
    psim2 = 0
    psim3 = 0
    if m >= 0:
        suss = reduce(su,f1)
        psim1 = 1.0 * 2 ** m * suss / n - n
    if m >= 1:
        suss = reduce(su,f2)
        psim2 = 1.0 * 2 ** (m - 1) * suss / n - n
    if m >= 2:
        suss = reduce(su,f3)
        psim3 = 1.0 * 2 ** (m - 2) * suss / n - n
    d1 = psim1-psim2
    d2 = psim1-2 * psim2 + psim3
    pval1 = spc.gammaincc(2 ** (m - 2), d1 / 2.0)
    pval2 = spc.gammaincc(2 ** (m - 3), d2 / 2.0)
    return [pval1, pval2]


def cumultativesumstest(binin):

    n = len(binin)
    ss = [int(el) for el in binin]
    sc = map(sumi, ss)
```

```
    cs = np.cumsum(sc)
    z = max(abs(cs))
    ra = 0
    start = int(np.floor(0.25 * np.floor(-n / z) + 1))
    stop = int(np.floor(0.25 * np.floor(n / z) - 1))
    pv1 = []
    for k in xrange(start, stop + 1):
        pv1.append(sst.norm.cdf((4 * k + 1) * z / np.sqrt(n)) -
        sst.norm.cdf((4 * k - 1) * z / np.sqrt(n)))
    start = int(np.floor(0.25 * np.floor(-n / z - 3)))
    stop = int(np.floor(0.25 * np.floor(n / z) - 1))
    pv2 = []
    for k in xrange(start, stop + 1):
        pv2.append(sst.norm.cdf((4 * k + 3) * z / np.sqrt(n)) -
        sst.norm.cdf((4 * k + 1) * z / np.sqrt(n)))
    pval = 1
    pval -= reduce(su, pv1)
    pval += reduce(su, pv2)

    return pval

def cumultativesumstestreverse(binin):

    pval=cumultativesumstest(binin[::-1])
    return pval

def pik(k,x):
    if k==0:
        out=1-1.0/(2*np.abs(x))
    elif k>=5:
        out=(1.0/(2*np.abs(x)))*(1-1.0/(2*np.abs(x)))**4
    else:
        out=(1.0/(4*x*x))*(1-1.0/(2*np.abs(x)))**(k-1)
    return out
```

```
def randomexcursionstest(binin):

    xvals=[-4, -3, -2, -1, 1, 2, 3, 4]
    ss = [int(el) for el in binin]
    sc = map(sumi,ss)
    cumsum = np.cumsum(sc)
    cumsum = np.append(cumsum,0)
    cumsum = np.append(0,cumsum)
    posi=np.where(cumsum==0)[0]
    cycles=([cumsum[posi[x]:posi[x+1]+1] for x in xrange(len(posi)-1)])
    j=len(cycles)
    sct=[]
    for ii in cycles:
        sct.append(([len(np.where(ii==xx)[0]) for xx in xvals]))
    sct=np.transpose(np.clip(sct,0,5))
    su=[]
    for ii in xrange(6):
        su.append([(xx==ii).sum() for xx in sct])
    su=np.transpose(su)
    pikt=([([pik(uu,xx) for uu in xrange(6)]) for xx in xvals])
    # chitab=1.0*((su-j*pikt)**2)/(j*pikt)
    chitab=np.sum(1.0*(np.array(su)-j*np.array(pikt))**2
            /(j*np.array(pikt)),axis=1)
    pval=([spc.gammaincc(2.5,cs/2.0) for cs in chitab])
    return pval

def getfreq(linn, nu):
    val = 0
    for (x, y) in linn:
        if x == nu:
            val = y
    return val

def randomexcursionsvarianttest(binin):
```

```
    ss = [int(el) for el in binin]
    sc = map(sumi, ss)
    cs = np.cumsum(sc)
    li = []
    for xs in sorted(set(cs)):
        if np.abs(xs) <= 9:
            li.append([xs, len(np.where(cs == xs)[0])])
    j = getfreq(li, 0) + 1
    pval = []
    for xs in xrange(-9, 9 + 1):
        if not xs == 0:
            # pval.append([xs, spc.erfc(np.abs(getfreq(li, xs) - j) /
            np.sqrt(2 * j * (4 * np.abs(xs) - 2)))])
            pval.append(spc.erfc(np.abs(getfreq(li, xs) - j) /
            np.sqrt(2 * j * (4 * np.abs(xs) - 2))))
    return pval

def aproximateentropytest(binin, m=10):

    n = len(binin)
    f1a = [(binin + binin[0:m - 1:])[xs:m + xs:] for xs in xrange(n)]
    f1 = [[xs, f1a.count(xs)] for xs in sorted(set(f1a))]
    f2a = [(binin + binin[0:m:])[xs:m + 1 + xs:] for xs in xrange(n)]
    f2 = [[xs, f2a.count(xs)] for xs in sorted(set(f2a))]
    c1 = [1.0 * f1[xs][1] / n for xs in xrange(len(f1))]
    c2 = [1.0 * f2[xs][1] / n for xs in xrange(len(f2))]
    phi1 = reduce(su, map(logo, c1))
    phi2 = reduce(su, map(logo, c2))
    apen = phi1 - phi2
    chisqr = 2.0 * n * (np.log(2) - apen)
    pval = spc.gammaincc(2 ** (m - 1), chisqr / 2.0)
    return pval

def matrank(mat): ## old function, does not work as advertized -
gives the matrix rank, but not binary
```

```
    u, s, v = np.linalg.svd(mat)
    rank = np.sum(s > 1e-10)
    return rank

def mrank(matrix): # matrix rank as defined in the NIST specification
    m=len(matrix)
    leni=len(matrix[0])
    def proc(mat):
        for i in xrange(m):
            if mat[i][i]==0:
                for j in xrange(i+1,m):
                    if mat[j][i]==1:
                        mat[j],mat[i]=mat[i],mat[j]
                        break
            if mat[i][i]==1:
                for j in xrange(i+1,m):
                    if mat[j][i]==1: mat[j]=[mat[i][x]^mat[j][x]
                    for x in xrange(leni)]
        return mat
    maa=proc(matrix)
    maa.reverse()
    mu=[i[::-1] for i in maa]
    muu=proc(mu)
    ra=np.sum(np.sign([xx.sum() for xx in np.array(mu)]))
    return ra

def binarymatrixranktest(binin,m=32,q=32):

    p1 = 1.0
    for x in xrange(1,50): p1*=1-(1.0/(2**x))
    p2 = 2*p1
    p3 = 1-p1-p2;
    n=len(binin)
    u=[int(el) for el in binin] # the input string as numbers,
                                to generate the dot product
```

```
    f1a = [u[xs*m:xs*m+m:] for xs in xrange(n/m)]
    n=len(f1a)
    f2a = [f1a[xs*q:xs*q+q:] for xs in xrange(n/q)]
    # r=map(matrank,f2a)
    r=map(mrank,f2a)
    n=len(r)
    fm=r.count(m);
    fm1=r.count(m-1);
    chisqr=((fm-p1*n)**2)/(p1*n)+((fm1-p2*n)**2)/(p2*n)+
           ((n-fm-fm1-p3*n)**2)/(p3*n);
    pval=np.exp(-0.5*chisqr)
    return pval

def lincomplex(binin):
    lenn=len(binin)
    c=b=np.zeros(lenn)
    c[0]=b[0]=1
    l=0
    m=-1
    n=0
    u=[int(el) for el in binin] # the input string as numbers, to generate
                                the dot product
    p=99
    while n<lenn:
        v=u[(n-l):n] # was n-l..n-1
        v.reverse()
        cc=c[1:l+1] # was 2..l+1
        d=(u[n]+np.dot(v,cc))%2
        if d==1:
            tmp=c
            p=np.zeros(lenn)
            for i in xrange(0,l): # was 1..l+1
                 if b[i]==1:
                     p[i+n-m]=1
            c=(c+p)%2;
```

```
            if l<=0.5*n: # was if 2l <= n
                 l=n+1-l
                 m=n
                 b=tmp
        n+=1
    return l

# test 2.10
def linearcomplexitytest(binin,m=500):

    k = 6
    pi = [0.01047, 0.03125, 0.125, 0.5, 0.25, 0.0625, 0.020833]
    avg = 0.5*m + (1.0/36)*(9 + (-1)**(m + 1)) - (m/3.0 + 2.0/9)/2**m
    blocks = stringpart(binin, m)
    bign = len(blocks)
    lc = ([lincomplex(chunk) for chunk in blocks])
    t = ([-1.0*(((-1)**m)*(chunk-avg)+2.0/9) for chunk in lc])
    vg=np.histogram(t,bins=[-9999999999,-2.5,-1.5,-0.5,0.5,1.5,2.5,
       9999999999])[0][::-1]
    im=([((vg[ii]-bign*pi[ii])**2)/(bign*pi[ii]) for ii in xrange(7)])
    chisqr=reduce(su,im)
    pval=spc.gammaincc(k/2.0,chisqr/2.0)
    return pval

def testall(bits):
    print 'Length:\t\t\t\t\t', len(bits)
    print
    print 'monobitfrequencytest\t\t\t', monobitfrequencytest(bits)
    print 'blockfrequencytest\t\t\t', blockfrequencytest(bits, 3)
    print 'runstest\t\t\t\t', runstest(bits)
    print 'spectraltest\t\t\t\t', spectraltest(bits)
    print 'nonoverlappingtemplatematching\t\t',
    nonoverlappingtemplatematchingtest(bits, '1001', 10)
    print 'overlapingtemplatematching\t\t',
           overlappingtemplatematchingtest(bits, '100', 12, 5)
```

```
print 'serialtest\t\t\t\t', serialtest(bits, 10)
print 'cumulativesumstest\t\t\t', cumultativesumstest(bits)
print 'aproximateentropytest\t\t\t', aproximateentropytest(bits, 4)
print 'randomexcursionsvarianttest\t\t',
       randomexcursionsvarianttest(bits)
print "linearcomplexitytest\t\t\t",linearcomplexitytest(bits,10)
print "binarymatrixranktest\t\t\t",binarymatrixranktest(bits,3,4)
print "lempelzivcompressiontest\t\t",lempelzivcompressiontest(bits)
print "longestrunones10000\t\t",longestrunones10000(bits)
print "maurersuniversalstatistictest\t\t",
       maurersuniversalstatistictest(bits,12,5)
print "randomexcursionstest\t\t\t",randomexcursionstest(bits)
return
```

THIS PAGE INTENTIONALLY LEFT BLANK

List of References

[1] F. A. Petitcolas, "Kerckhoffs' Principle," in *Encyclopedia of Cryptography and Security*. New York, Springer, 2011, pp. 675–675.

[2] R. V. Shankar, "Shannon's Theory of Cryptography," Mar 1997. [Online]. Available: http://www.cse.iitm.ac.in/~theory/tcslab/cryptpage/report1.pdf

[3] B. Sunar, W. J. Martin, and D. R. Stinson, "A Provably Secure True Random Number Generator with Built-In Tolerance to Active Attacks," *IEEE Transactions on Computers*, vol. 56, no. 1, pp. 109–119, 2007.

[4] V. Fischer and M. Drutarovskỳ, "True random number generator embedded in reconfigurable hardware," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Heidelberg: Springer, 2002, pp. 415–430.

[5] P. Junod, "Cryptographic Secure Pseudo-Random Bits Generation: The Blum-Blum-Shub Generator," unpublished manuscript. 1999. [Online]. Available: https://crypto.junod.info/bbs.pdf

[6] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, and E. Barker, "A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications," National Institute of Standards and Technology, Gaithersburg, MD, Special Pub. 800-22 Rev. 1a, 2010.

[7] L. Blum, M. Blum, and M. Shub, "A Simple Unpredictable Pseudo-Random Number Generator," *SIAM Journal on Computing*, vol. 15, no. 2, pp. 364–383, 1986.

[8] A. Sidorenko and B. Schoenmakers, "Concrete Security of the Blum-Blum-Shub Pseudorandom Generator," in *IMA International Conference on Cryptography and Coding*. Eindhoven: Springer, 2005, pp. 355–375.

[9] J. Booher, "*Square Roots in Finite Fields and Quadratic Nonresidues*," Palo Alto, CA: Stanford University, 2012.

[10] C. Ding, "Blum-Blum-Shub Generator," *Electronics Letters*, vol. 33, no. 8, p. 677, 1997.

[11] U. V. Vazirani and V. V. Vazirani, "Efficient and Secure Pseudo-Random Number Generation," in *Workshop on the Theory and Application of Cryptographic Techniques*. Heidelberg: Springer, 1984, pp. 193–202.

[12] D. Manandhar, personal communication, March 21, 2016.

[13] S. Tardieu, “Prime Generation in Python,” 2010 (accessed February 3, 2016). [Online]. Available: https://github.com/VSpike/BBS/blob/master/primes.py

[14] I. Gerhardt, “Random Number Testing in Python,” 2009 (accessed February 3, 2016). [Online]. Available: https://gerhardt.ch/random.php

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California